software/FPGA co-design for Edge-computing : Promoting object-oriented design

Xuan Sang Le

UBO
université de bretagne occidentale

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE BRETAGNE OCCIDENTALE
sous le sceau de l'Université Bretagne Loire

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE
Mention : Informatique

École Doctoral Santé, Information, Communication, Mathématique, Matière

présentée par
Xuan Sang LE

préparée à
École Mines-Télécom, IMT Lille Douai *et*
Lab-STICC UMR CNRS 6285,
École Nationale Supérieure de Techniques Avancées de Bretagne

Software/FPGA Co-design for Edge-computing: Promoting Object-oriented Design

Thèse soutenue le 31 Mai 2017
devant le jury composé de:

Olivier ROMAIN
Professeur, Université de Cergy-Pontoise/ Rapporteur

Anne ETIEN
Maître de conférences HDR, Polytech Lille/ Rapporteur

Thomas LEDOUX
Maître de conférences, IMT Atlantique/ Examinateur

Ahcene BOUNCEUR
Maître de conférences HDR, Université de Bretagne Occidentale/ Examinateur

Loïc LAGADEC
Professeur, ENSTA Bretagne / Directeur

Noury BOURAQADI
Professeur,Mines-Télécom, IMT Lille Douai/ Co-directeur

IMT Lille Douai
École Mines-Télécom
IMT-Université de Lille

ENSTA Bretagne

Lab-STICC

Co-advisors:
Loic Lagadec &
Noury Bouraqadi

Xuan Sang LE

Abstract

Cloud computing is often the most referenced computational model for Internet of Things. This model adopts a centralized architecture where all sensor data is stored and processed in a sole location. Despite of many advantages, this architecture suffers from a low scalability while the available data on the network is continuously increasing. It is worth noting that, currently, more than 50% internet connections are between things. This can lead to the reliability problem in realtime and latency-sensitive applications. Edge-computing, which is based on a decentralized architecture, is known as a solution for this emerging problem by: (1) reinforcing the equipment at the edge (things) of the network and (2) pushing the data processing to the edge.

Edge-centric computing requires sensors nodes with more software capability and processing power while, like any embedded systems, being constrained by energy consumption. Hybrid hardware systems consisting of FPGA and processor offer a good trade-off for this requirement. FPGAs are known to enable parallel and fast computation within a low energy budget. The coupled processor provides a flexible software environment for edge-centric nodes.

Applications design for such hybrid network/software/hardware (SW/HW) system always remains a challenged task. It covers a large domain of system level design from high level software to low-level hardware (FPGA). This results in a complex system design flow and involves the use of tools from different engineering domains. A common solution is to propose a heterogeneous design environment which combining/integrating these tools together. However, the heterogeneous nature of this approach can pose the reliability problem when it comes to data exchanges between tools.

Our motivation is to propose a homogeneous design methodology and environment for such system. We study the application of a modern design methodology, in particular object-oriented design (OOD), to the field of embedded systems. Our choice of OOD is motivated by the proven productivity of this methodology for the development of software systems. In the context of this thesis, we aim at using OOD to develop a homogeneous design environment for edge-centric systems. Our approach addresses three design concerns: (1) hardware design, where object-oriented principles and design patterns are used to improve the reusability, adaptability, and extensibility of the hardware system. (2) hardware / software co-design, for which we propose to use OOD to abstract the SW/HW integration and the communication that encourages the system modularity and flexibility. (3) middleware design for Edge Computing. We rely on a centralized development environment for distributed applications, while the middleware facilitates the integration of the peripheral nodes in the network, and allows automatic remote reconfiguration. Ultimately, our solution offers software flexibility for the implementation of complex distributed algorithms, complemented by the full exploitation of FPGAs performance. These are placed in the nodes, as close as possible to the acquisition of the data by the sensors, in order to deploy a first effective intensive treatment.

Co-directeurs:
Loic Lagadec &
Noury Bouraqadi

Xuan Sang LE

Résumé

L'informatique en nuage (cloud computing) est souvent le modèle de calcul le plus référencé pour l'internet des objets (Internet of Things). Ce modèle adopte une architecture où toutes les données de capteur sont stockées et traitées de façon centralisée. Malgré de nombreux avantages, cette architecture souffre d'une faible évolutivité alors même que les données disponibles sur le réseau sont en constante augmentation. Il est à noter que, déjà actuellement, plus de 50 % des connexions sur Internet sont inter objets. Cela peut engendrer un problème de fiabilité dans les applications temps réel. Le calcul en périphérie (Edge computing) qui est basé sur une architecture décentralisée, est connue comme une solution pour ce problème émergent en: (1) renforçant l'équipement au bord du réseau et (2) poussant le traitement des données vers le bord.

Le calcul en périphérie nécessite des nœuds de capteurs dotés d'une plus grande capacité logicielle et d'une plus grande puissance de traitement, bien que contraints en consommation d'énergie. Les systèmes matériels hybrides constitués de FPGAs et de processeurs offrent un bon compromis pour cette exigence. Les FPGAs sont connus pour permettre des calculs exhibant un parallélisme spatial, aussi que pour leur rapidité, tout en respectant un budget énergétique limité. Coupler un processeur au FPGA pour former un noeud garantit de disposer d'un environnement logiciel flexible pour ce noeud.

La conception d'applications pour ce type de systèmes hybrides (réseau/logiciel/matériel) reste toujours une tâche difficile. Elle couvre un vaste domaine d'expertise allant du logiciel de haut niveau au matériel de bas niveau (FPGA). Il en résulte un flux de conception de système complexe, qui implique l'utilisation d'outils issus de différents domaines d'ingénierie. Une solution commune est de proposer un environnement de conception hétérogène qui combine/intègre l'ensemble de ces outils. Cependant, l'hétérogénéité intrinsèque de cette approche peut compromettre la fiabilité du système lors des échanges de données entre les outils.

L'objectif de ce travail est de proposer une méthodologie et un environnement de conception homogène pour un tel système. Cela repose sur l'application d'une méthodologie de conception moderne, en particulier la conception orientée objet (OOD), au domaine des systèmes embarqués. Notre choix de OOD est motivé par la productivité avérée de cette méthodologie pour le développement des systèmes logiciels. Dans le cadre de cette thèse, nous visons à utiliser OOD pour développer un environnement de conception homogène pour les systèmes de type Edge Computing. Notre approche aborde trois problèmes de conception: (1) la conception matérielle, où les principes orientés objet et les patrons de conception sont utilisés pour améliorer la réutilisation, l'adaptabilité et l'extensibilité du système matériel. (2) la co-conception matériel/logiciel, pour laquelle nous proposons une utilisation de OOD afin d'abstraire l'intégration et la communication entre matériel et logiciel, ce qui encourage la modularité et la flexibilité du système. (3) la conception d'un intergiciel pour l'Edge Computing. Ainsi il est possible de reposer sur un environnement de développement centralisé des applications distribuées, tandis ce que l'intergiciel facilite l'intégration des noeuds périphériques dans le réseau, et en permet la reconfiguration automatique à distance. Au final, notre solution offre une flexibilité logicielle pour la mise en oeuvre d'algorithmes distribués complexes, et permet la pleine exploitation des performances des FPGAs. Ceux ci sont placés dans les noeuds, au plus près de l'acquisition des données par les capteurs, pour déployer un premier traitement intensif efficace.

To my daughter, Uyen Nhi

Contents

Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisors Prof. Noury Bouraqadi and Prof. Loic Lagadec for the continuous support of my Ph.D study and related research, for their patience, motivation, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having better advisors and mentors for my Ph.D study.

My sincere thanks also goes to my supervisors Dr. Luc Fabresse, Dr. Jean-Christophe Le Lann, and Dr. Jannik Laval, for their kindness, their availability and their advices on techincal and scientific aspects during this thesis. Without their precious support it would not be possible to conduct this research.

Besides my advisors and supervisors, I would like to thank the rest of my thesis committee: Prof. Olivier Romain, Dr. Anne Etien, Dr. Thomas Ledoux and Dr. Ahcene Bounceur, for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

I thank my fellow labmates at IMT Lille Douai and ENSTA Bretagne with whom i have pleasure to work and for all the fun we have had in the last three years.

Last but not the least, I would like to thank my family: my wife, Phuong TRAN and to my daughter, Uyen Nhi, for supporting me spiritually throughout writing this thesis and my life in general.

Xuan Sang LE

List of Figures

Listings

Glossary

API Application Programming Interface. iv, x, xi, 2, 8, 14, 20, 30, 31, 35, 48, 59, 62, 65–71, *75*, *76*, *79*, 81–85, 89, 98

AST Abstract Syntax Tree. 99

AXI Advanced eXtensible Interface. 62

BLIF Berkeley Logic Interchange Format. 49

CAD Computer-aided design. 31, 32, 48, *57*, 98, 99

CLB Configurable Logic Block(s). 4

CORBA Common Object Request Broker Architecture. 87

CPS Cyber-physical Systems. 3

DOA Distributed Objects API. 83–85, 87, 94, 95, 98, 99

DSL Domain Specific Language. iii, 29–31, 35, 37, 38, 49, *57*, 97

EC Edge Computing. 3, 4, 10, 11

EDA Electronic Design Automation. 12, *17*, 18

FIR Finite impulse response. ix, xi, 35, 36, 48

FPGA Field Programmable Gate Array. iii, ix, 1, 4, 5, *7*, 10, 11, 13, 15, 21, 25–29, 31, 32, 35, 52, *55*, 60–62, 64–*75*, 80–82, 85, 87–91, 97–100

FSM Finite State Machine. 18, 20, 26, *27*

FSMD Finite State Machine with Datapath. *17*

GPIO General Purpose Input/Output. *72*

GUI Graphic User Interface. ix, 18, 31, *55*, *56*, 99

HDL Hardware Description Language. 12–14, *17*, 18, 20–22, 24, 25, *27*, 28, 30, 31, 35, 51, 52

HLS High Level Synthesis. 13, 24, 26, 30, 60

HTTP Hypertext Transfer Protocol. 82, 87, 90

HW Hardware. iii, iv, ix, x, 5–8, 12, 13, 16, 19, 21–31, 34, 35, 38, 39, 45, 51, 59, 60, 65, *79*–82, 85, 86, 97–99

I2C Inter-Integrated Circuit. xi, 45, 46

IEEE Institute of Electrical and Electronics Engineers. 17, 32, 48

IoT Internet of Things. iii, 1–4, 9–11, 27, 28, 97, 100

IP (1) "Internet Protocol" in the context of networking, in case of hardware design, the term means (2) "Intellectual Property". ix, x, 8, 10–13, 17–22, 24, 25, 27, 30, 31, 50–52, 57, 63–65, 79–82, 95, 97, 98

JSON Javascript Object Notation. 82, 84

MARTE Modelling and Analysis of Real-time and Embedded Systems. 20, 21, 26

MDE Model Driven Engineering. ix, 8, 13, 14, 16, 18, 20–24, 26, 27, 30

MOF Meta Object Facility. 15

OMG Object Management Group. 15

OO Object Oriented. iv, 20–22, 29, 38, 40

OOD "Object Oriented Design". iv, x, 6–8, 18, 20–22, 24, 25, 27–31, 34, 35, 40, 63, 64, 97–99

OOL Object Oriented Language. 20

OoRC Objectification of Reconfigurable Circuits. iii, 8, 29–35, 47–53, 55, 57, 61, 81

PCLe (Peripheral Component Interconnect Express. 60

REST Representational state transfer. 2, 81, 82, 84, 87, 89, 90

ROS Robot Operating System. iv, xi, 25, 59, 71, 75, 76

RTL Register Transfer Level. iii, 7, 8, 12, 13, 17, 27, 29–32, 48, 91, 97

SN Sensor Network(s). 2, 3, 10, 11, 80, 87, 100

SOAP Simple Object Access Protocol. 2

SoC System on a Chip. 19–21, 26, 60

SW Software. iv, ix, x, 5–8, 11, 12, 16, 18, 19, 24–28, 59–61, 65, 79–82, 85, 86, 97–99

UART Universal asynchronous receiver/transmitter. xi, 43, 45, 46

UML Unified Modelling Language. iii, 9, 14, 20–22, 25–27, 32, 35, 99

VHDL Very High Speed Integrated Circuit Hardware Description Language. iv, ix–xi, 5, 8, 12, 18, 20, 21, 25, 29, 31–35, 38, 39, 47–51, 55, 57, 64, 67, 68, 72–74, 81, 89, 91, 97, 99

VM Virtual Machine. 81, 82, 85, 87, 90, 98, 99

WIEM Wireless External Interface Module. 60, 62, 67, 68, 88

XML Extensible Markup Language. 17, 18, 82

1 Introduction

Contents

1.1 Context: Internet of Thing, Edge computing and FPGA

1.1.1 Internet of Things

Internet of things (IoT) is a concept increasingly supported by various stakeholders and market forces. It is foreseen to be a world-wide network of interconnected devices or objects (things) through wired and wireless connections [VF13]. The network provides a unique addressing scheme and creates a pervasive environment where a person can interact anytime with the digital and physical worlds. The primary goal is to enable things to be connected anytime, anywhere and with anything or persons using existing network infrastructure. Objects can identify themselves and have seamless intelligence for context decision making. IoT can be considered as the next evolution of the internet [Eva11] and has many potential applications, especially in smart systems such as healthcare, Smart Cities, Smart Grids, Smart Cars and mobility, Smart Homes and Assisted Living, Smart Industries, Public safety, Energy & environmental protection, Agriculture, etc.

1.1.1.1 Resource for IoT

The existing internet infrastructure is the primary resource for IoT. Most IoT devices are IP-enabled* and are able to easily participate to the internet. Web services –such as REST, SOAP, etc.– are well established mechanisms for the communication between IoT nodes. The advantage of web services is that they are general purpose and thus, can be easily integrated to others systems (e.g. IoT systems) that are built on standard (and general purpose) IT components. The passing of IP to IPv6 allows an unrestricted address scheme and enables a large-scale of things.

At the physical level (sensor, actuator, etc.), where the devices (things) actually interact with the real world, Sensor Networks (SN) are an important resource for IoT architecture. IoT can be considered as an evolution [MPV11] of SN at an internet-scale, thus existing work on SN can be adapted to IoT. SN offer a virtual layer where the digital system can communicate with the physical environment. Typically, a SN architecture consist of 4 main concepts [CF14]:

- Network components: elements that enable connectivity within the network, connect an application platform at one end of the network with one or more actual physical devices. These components correspond to the concepts of *node*, *gateway*, *relay*, *sensor* and *actuator*.
- Hardware platforms: The hardware requirement necessary for sensor/actuator nodes, computational/functional nodes (relay, server).
- Middlewares: software stack create an virtual layer between application and the physical world. They aim at operating, monitoring and managing the sensor network. They are often generic and flexible to different application scenario.
- Topology: describes how the SN is organized. The most common network topologies used in SN are star, tree, mesh or hybrid networks that combine the other ones. In the context of IoT, any kind of IP-based topologies can be applied.

At the processing level, *cloud computing* is currently the most referenced computational model for IoT [BdDPP16]. *Cloud computing* adopts a centralized architecture with large virtual ability of storage and processing power. It is defined [MG11] as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction...". IoT can benefit from this model for sensor data collecting, processing and decision making. The cloud can expose different services for sensor data usage (e.g. API, web services). It creates a virtual layer between things and applications, thus hiding the complexity of tasks like system development, integration and management, etc.

1.1.1.2 Potential Problem of current IoT configuration

For the last decades, we have seen an important increase of IoT devices. A research in [Fri13] (2013) shows that, in 2011, there were more than 15 billion things on the internet, over 50% internet connections are between or with things. By 2020, this number can go up to more than 30 billion things and

*Internet Protocol

over 200 billion intermittent connections are forecasted. All the connections are based on the client-cloud model. Sensor data such as thermostats, surveillance cameras, healthcare system measurement, etc. are centralized to the cloud. This is not always obvious since IoT devices have different radical characteristics. Some are occasional accessed while others require realtime always-on connections. Connection bandwidth[†] is another crucial factor, many devices are satisfied with low bandwidth connections while others need a high bandwidth for data transfer. This is often referred as the Big-data problem [ZE+11].

This centralized infrastructure of cloud suffers from a low scalability while the available data is continuously increasing. In real-time IoT systems this can lead to latency problems. That is, a large amount of IoT nodes may cause a work overhead to the system due to an intensive quantity of data to be transferred over the network. This costs network resources and degrades the response time of the system. The value for decision making of sensor data may be lost while it is traveling across the network. For data processing, although multi-core processors (on the cloud) are powerful enough for mass processing, it is not guaranty that they will be suitable for future calculation requirements of IoT.

Sensor networks, the base of IoT, are often considered as networks of simple devices with limited performance. These devices are only used for data acquisition and transmission. This restricted vision leads to simple communication-centric middleware models. While these devices and middleware respond well the requirements of SN, they may be not suitable at the scale of IoT. IoT systems, especially in smart system, may need more powerful devices with the ability to perform some local computations and decision making.

1.1.2 Edge Computing for Cyber-physical Systems

Cyber-physical systems (CPS) are integrations of computation and physical processes [Lee08]. It can be considered as a merger of embedded systems –a composition of standalone computing elements – and sensor network. CPS shares the same basic architecture with IoT. Nevertheless, it presents a higher combination and coordination between physical and computational elements [RHTO15]. The integration of software-intensive embedded systems and internet communication into CPS is considered to be the next revolution of IoT. CPS provides the necessary infrastructure to deal with the Big-data problems.

"In many aspects of human activity, there has been a continuous struggle between the forces of centralization and decentralization..." [GLME+15]. While today internet activities are dominant by the cloud, solution for the emerging IoT problems (as presented) requires a new evolution for the computational model. The network topology and the computing resource distribution need to be revised, in order to exploit the powerful features of IoT. The combination of Edge computing (EC) and CPS can be considered as a solution for these problems by: (1) migrating from a centralized architecture (cloud) to a distributed computing architecture; (2) strengthening the equipment at the edge of the network, where sensors are deployed and (3) performing analytics and knowledge generation at the source of the data.

[†]Today, two thirds of Internet traffic are dedicated to image transmission, this can go up to 80% in the next years

Lopez et al. [GLME+15] and *Faure et al.* [FFH+11b, FFH+11a] share a human-centric vision on EC in which they consider the important role of human in the edge devices. This vision shows a global view of edge-centric computing with different interesting discussions about: Trust, Privacy, Control, Intelligence and Proximity. My vision here, in this thesis, is limited in the context of IoT and shares some similarities with one presented by *Bonomi et al.* [BMZA12], where EC extends the cloud computing to the edge of the network. EC considers a node-oriented view of the internet. This architecture consists of data center and clouds at the core and surrounding by nodes with small web server and content-distributed network. At the edge of the network, some distinguishing characteristics can be found:

- *Low latency and proximity*: by pushing the data processing to the edge of the network, a large amount of raw sensor data can be processed locally in order to produce a compact and rich information before delivery to the centralized point. This reduces the network traffic and thus allows a fast response time. It also adds the possibility of filtering sensor data before delivery. EC promotes also the distributed communication between closed nodes rather than using a far-away central point (cloud).
- *Scalability*: EC allows large-scale sensor network with large number of nodes since the network overload are distributed to the edge.
- *Realtime interaction*: EC is suitable for realtime IoT applications by allowing fast response time between things and application (over the network).
- *Heterogeneity*: different form factor of edge-devices can be easily deployed in wide-variety of EC environment.
- *Intelligence and Control are on the edges*: nowadays, hardware devices become smaller and more powerful while being cheaper. Thus, IoT devices have more processing and storage capacity. This implies that edge-centric nodes are more capable and can make local decision and control what to do with sensor data.

Edge-centric nodes are processing-centric, thus, requires more processing capability. However, as regular embedded systems, they are constrained by power consumption. There are always the struggle to balance the performance/energy ratio on these systems. Moreover, as complex processing units, edge-centric nodes need an evolutive and flexible architecture to boost the application development productivity.

1.1.3 Using FPGAs for Edge Computing in IoT: Benefits and Challenges

1.1.3.1 FPGAs

FPGAs [Xil] –stands for Field Programmable Gate Array– are integrated circuits designed to be reconfigured by designers. They were first invented by Xilinx in 1985 and have been increasingly grown up since then. Xilinx and Altera (now acquired by Intel) are two vendors that have a crucial role in the development of FPGAs. These devices are based on a matrix of configurable logic blocks (CLBs) connected by programmable interconnects (switches). Thus FPGAs can be reconfigured to fit different application contexts after manufacturing. This can be done by specifying the functionality of each CLB and then wiring them together (using configurable switches). Traditionally, FPGA configurations are

described using Hardware Description Languages such as VHDL or Verilog. These descriptions are synthesized, using dedicated synthesis tools, into binary form (bitstream), that configures the device.

FPGAs are used in wide range of applications such as Aerospace,Automotive, Data Center, Medical, Security, Image and Signal processing, etc. Advantageous characteristics of FPGAs are: (1) *Reconfigurability and Flexibility*: FPGAs are reconfigurable and can implement easily tailored circuits for different application contexts; (2) *Performance*: FPGAs enable parallel and fast computations, thus are suitable for acceleration purpose; (3)*Energy consummation*: direct parallel hardware execution of tasks avoids the overhead problem of traditional software system such as processors and thus is energy-friendly; (4) *Maintainability*: FPGAs circuits can be easily reconfigured when performance improvements or bug fix are available; (5) *Reliability*: FPGA circuits are a real hardware implementation of tasks, they minimize the reliability problem in comparison to software on processors since tasks are executed in parallel and have their own deterministic hardware resource.

1.1.3.2 FPGAs and Edge-computing

As stated, in edge-computing, edge-centric nodes' functionality often exceeds simple data collection and embeds more complex features. These nodes require more processing capabilities while keeping power consumption low. Reconfigurable architectures such as FPGAs are known to enable parallel and fast computations within a low energy budget, hence can play a crucial role in edge-centric nodes [DLPBT12]. FPGAs add more processing ability to the nodes and thus allow to locally perform more sophisticated tasks. Furthermore, using FPGAs improves the flexibility of the node's hardware. That is, the hardware can be simply tailored to adapt to different kinds of sensors.

Today FPGA devices often come in a hybrid form factor consisting of an FPGA coupled with an embedded processor (e.g. ARM). This kind of devices is ideal for building edge-centric nodes since: (1) they offer the hardware acceleration (FPGA) while (2) providing a flexible and powerful software environment (processor) for complex middleware and applications (IP-stack, web services).

1.1.3.3 Challenges

Digital Hardware (FPGAs) design is always a complicate task. The design process requires a specific knowledge which remains a challenge for developers and usually results in a loss of productivity. Especially, when FPGAs are used together with advanced software systems (edge-centric middleware, web services, distributed content network, etc.), the problem of HW/SW co-design (detailed in section 2.4) becomes a real challenge. There is a need for a dedicated environment to support efficient development and deployment of such hybrid systems into the network. The environment should address to the following criteria:

- *Generalization and Heterogeneity*: since we target a network of nodes rather than a single device, the design environment (both SW and HW) should be generic enough to deal with different edge-centric devices (FPGA/processor). A standardized environment also helps applying design constraints on the system and thus better support system scalability.
- *Integrability and Interoperability*: the environment should promote the integration of: (1) SW and HW parts in an individual node and (2) the integration of each node to the over-all net-

work. A common communication protocol is needed for the SW/HW interfacing as well as the distribution of content over the network.

- *Reconfigurability and maintainability* : a network may contain hundreds to thousands of hybrid nodes, manual deployment, reconfiguration, or maintenance of each node is an important issue. An automatic mechanism is mandatory for remote (over-the-network) deployment and configuration of nodes, both on SW and HW.

- *Distributed computing*: Edge computing relies on distributed computing to push the processing to the edge. Therefore, middleware should promote the development and deployment of distributed algorithm/application over the network.

These challenges cover a large domain of system level design from high level software to low level hardware. On the one hand, we rely on an high-end edge-centric (based on web services) software environment for end user application development and deployment. On the other hand, we need to deal with low-level FPGA design and SW/HW integration. There is always the need of a unified design methodology and environment for closing this important gap between the two worlds.

1.2 Research Objectives and Contributions

1.2.1 Research Objectives

When looking at existing design methodologies for both SW and HW system, it is clear that software design methodologies, such as Object Oriented Design (OOD), are in advance of hardware, in term of *productivity*. Hardware design always remains a long and tedious process, especially when it comes to the SW/HW co-design problem. This result in a complex system design flow and involve the use of tools from different engineering domains. Common solutions tend to propose a heterogenous design environment that integrates/combines these tools togethers. However, the heterogeneous nature of this approach can pose the reliability problem when it comes to data exchanges between tools of different expertise domains (i.e. syntactical and semantics interoperability). Our motivation is to propose a homogeneous design methodology and environment for such system to minimize this problem.

This work studies the application of modern design methodologies, in particular the OOD, on embedded system design. Firstly, we aim at profiting object oriented principles to hardware design for a more productive HW design environment. This latter complements the traditional HW design methodologies by promoting the concepts of *Generalization*, *Generation*, *Standardization* and *Separation of concerns*. Therefore it allows better system *Reusability*, *Maintainability*, and *Extendability*.

Secondly, we explore the use of OOD to produce a uniform methodology and environment for SW/HW co-design. The largest gain of OOD here is the ability to abstract the SW/HW integration process in an implementation-independent way. Hence, the interfacing gap can be automatized at certain level by some correct-by-construction and automatic generation techniques. This is important seeing that closing SW/HW gap remains always a complex, time-consuming, error-prone and less-contributive task.

Last but not least, we want to position the proposed design approaches on the context of edge-centric computing where the target system becomes much more complicated with a network of hybrid

SW/HW nodes connected together. This forms a complex distributed environment and rises the question of how to efficiently manage, develop, deploy and maintain such system. A manual solution is not an option considering the network scale of the system. In this case, while the processing is decentralized (distributed), the development (both SW and HW), should be centralized to facilitate the management and maintenance. The deployment should be automatized to be able to handle a large scale of nodes. Such dedicated edge-centric environment is still missing in our perspective.

1.2.2 Contributions

The research towards this thesis has been partly included in the following publications (papers or poster, chronological order):

1. Xuan Sang LE, Loïc Lagadec, Luc Fabresse, Jannik Laval, and Noury Bouraqadi. From Smalltalk to Silicon: Towards a methodology to turn Smalltalk code into FPGA. In *IWST 14*, Cambridge, United Kingdom, August 2014

2. Xuan Sang LE, Loïc Lagadec, Luc Fabresse, Jannik Laval, and Noury Bouraqadi. A meta model supporting both hardware and smalltalk-based execution of fpga circuits. *IWST '15*, pages 6:1–6:14, 2015. IWST best paper award

3. Xuan Sang LE, Luc Fabresse, Jannik Laval, Jean-Christophe Le Lann, Loïc Lagadec, and Noury Bouraqadi. Dynamic distributed programming on reconfigurable ip-based smart sensor networks. Presented as poster at 11ème Colloque du GDR SoC-SiP, France, 2016

4. Xuan Sang LE, Luc Fabresse, Jannik Laval, Jean-Christophe Le Lann, Loïc Lagadec, and Noury Bouraqadi. Speeding Up Robot Control Software Through Seamless Integration With FPGA. In *SHARC '16: 11th National Conference on Software and Hardware Architectures for Robots Control*, Brest, France, 2016

5. Xuan Sang LE, Jean-Christophe Le Lann, Loïc Lagadec, Luc Fabresse, Noury Bouraqadi, and Jannik Laval. Cardin: An agile environment for edge computing on reconfigurable sensor networks. In *the proceedings of The 2016 International Conference on Computational Science and Computational Intelligence (CSCI'16)*, Las Vegas, Nevada, USA, 2016

The contribution presented in this thesis are designed and developed by the author. The software environment used for modeling and middleware, toolset building is based on Pharo Smalltalk [DZHC17] – an elegant object oriented language and environment. The research carried out for this thesis resulted in the following contributions (listed in the order they appear in this thesis):

1. Conceptual meta-model –baptized OoRC– that brings OOD concepts and principles to hardware design. The meta-model support modeling hardware system at two levels of granularity: (1) fine-grained level where it is used to specify FPGA circuits at RTL level; (2) coarse-grained level where OOD principles and design patterns are employed to abstract and modularize the hardware system.

2. Dedicated design environment (middleware and toolset) for SW/HW integration which is based on the OOD and platform-based design approaches. The essential is to close the SW/HW gap by abstracting the communication and provide automatic implementation (generation) of the interface (both SW and HW) depending on the application context.

3. Hardware architecture, middleware and toolset for edge-centric application development. The philosophy here is to (1) centralize the development of distributed (decentralized) application, (2) automatize the application deployment and (3) abstract the network communications.

OOD is the main design methodology employed throughout these contributions to produce a unique design environment and flow. It is used in conjunction with (1) Model-Driven Engineering (MDE) to model hardware system, (2) platform-based design for SW/HW co-design and (3) distributed programming to provide environment for distributed application development and deployment on the network.

1.3 Outline of the Thesis

Chapter 2 presents the state of the art of our research which covers the following domains: edge-computing, hardware design, and HW/SW co-design. Chapter 3 describes our OoRC meta-model at conceptual level. Both RTL-level modeling and system-level modeling are addressed. The chapter also describes how the meta-model handles the reuse of VHDL legacy IPs as regular object oriented models. Chapter 4 targets the SW/HW co-design problem with OoRCBridge, middleware and toolset dedicated for integrating FPGA devices in existing high-level software system. Concretely, the SW/HW communication is standardized, a mechanism for automatic interface generation is specified and an API for abstracting HW accessing is also provided. Chapter 5 puts it all together in the context of edge-computing. We present our dedicated distributed environment named CaRDIN. It has been developed for application building and deploying on such hybrid system. Finally, in chapter 6, conclusions are drawn and future work related to the OoRC meta-model and CaRDIN are discussed.

"Creativity is thinking up new things. Innovation is doing new things."

Theodore Levitt

2

State of the Art

Contents

This chapter covers an examination of existing research in the field of embedded system design methodologies and application of FPGA-based device in IoT and edge-centric computing. The first section enlightens the current use of SN and FPGA in the edge computing domain. We then analyze the hardware design gap by comparing different existing hardware design methodologies. Hardware/software co-design is another related important research field which will be discussed at the end of the chapter.

2.1 Edge Computing

2.1.1 Dedicated SN for Edge Computing

Edge centric architecture consists of a data center at the core surrounded by capable nodes with small web server constituting a content distribution network. Sensor Networks compliant with Internet protocol (IP) are good candidates for building such architecture. Unfortunately, most SNs solutions are non-IP-compatible SNs, thus have difficulty to participate to internet (missing of IP stack). In the context of EC, the integration of such SN to IP-based network requires the deployment of an extra layer at the edge of the two networks to link non-IP SN communications with internet communications [GRL+08, KBLK07]. Proposed approaches are focused on wrapping data coming from sensor sources for sharing and processing over the Internet. Those works provide heterogeneity out of SN. Communication and application-level code needs to be hand-programmed for each node. Recently, a new research trend has emerged in this area: the IP-based sensor networks, [PKGZ08, D+09], that offers a more natural and direct way to bring SN to the internet. These approaches assume that sensor nodes are powerful enough to implements an IP stack. The web services can be used on top of these systems and offer the compatibility between SN and standard IoT infrastructures. These works show an important advance in the area. They, however, provide only the infrastructure without further software API for the operations and interactions of the node.

Middleware plays a crucial role in SN, it defines an application platform that: (1) can be deployed in various application scenarios; (2) can handle the heterogeneous and distributed nature of the network and (3) promotes the integrability of the system. Diverse middleware solutions have been proposed for SN. Some approaches such as [YG02, MFHH05] consider the SN as a *virtual database* that can be queried through an SQL-like language. Others rely on *Mobile agent* approach in which applications are modular and each module can be distributed through the network [BHSS07, ORRM09]. The works in [MAK07, SCC+06] propose a *Virtual Machine* approach which is more general than Mobile Agents. They allow arbitrary code to run on sensor nodes and make the software independent from the hardware architecture. *Message oriented middleware* [KS09, CCD+09] is another possible solution for communication inside SN. These approach support sending and receiving messages between distributed systems via an event-driven mechanism (publish/subscribe service). These middle-wares could be a standalone architecture that runs directly on bare-metal hardware or could be built on top of an OS –such as TinyOS [LMP+05]– dedicated for sensor network. All of these middleware are used mainly for querying raw sensor data rather than support the development and deployment of complex applications on the nodes.

2.1.2 Using FPGAs for Edge Computing in IoT

Since the application of FPGAs for Edge Computing is a brand new topic, the state of the art is still limited in the context of IoT. However, we can consider the use of FPGA on SN as references, seeing that SN is the base of IoT and Edge Computing. Many researchers have explored the benefit of parallel processing and hardware reconfiguration in FPGA for prototyping sensor networks. Some approaches use only FPGA with [MR08, CSM08] or without [LSKT13, HRVG08] soft-core processor (e.g. NIOS) to build the sensor nodes. Those enable the flexible adaptation of hardware changes on sensor nodes, but do not offer the flexible reconfiguration that software approaches do, nor do they support remote reconfiguration of the node. Since the system is entirely implemented in FPGA, the software capacity of these nodes is limited that lead to the missing of an efficient middleware. Other works such as [Ber12, KPC+08, PRDC] address to these problems by proposing a non IP-based approach with an entire workflow to generate, remotely configure and reconfigure the FPGA. Theses work use a micro-controller (µC) to reconfigure the FPGA. Both software (µC) and hardware (FPGA) can be reprogrammed/reconfigured remotely from host via a wireless link (e.g. ZigBee).However, the use of µC still limits the software capability of the node. The development is entirely baremetal (µC/FPGA) which is specific and limits the reusability of the system. All of these approaches are non IP-based and are not compatible with an internet and edge-centric usage. Firstly, to support IoT, the IP-stack (and web services) must be implemented on µC or on FPGA which requires specific skills. Secondly, they are designed only for sensor data acquisition and thus are not suitable for edge-centric applications.

2.1.3 Discussion

Current proposed SN architectures are based on the cloud-computing model, where all sensor data are centralized and processed in a sole place. Sensor nodes are often simple devices with limited software capabilities and used for data acquisition and transmission. This results a very simple communication-based middleware. When passing to an edge computing model, these architectures and middlewares need to be revised. The IP protocol should be the base protocol for network communication. By reinforcing the edge of the network, edge-centric nodes are more powerful. This requires a new generation of middleware to be able to (1) efficiently exploit the hardware capability of the nodes, (2) provide a rich set SW features for building complex application.

FPGAs show interesting application in edge computing since they offer parallel and realtime processing ability to the node with a reasonable energy budget. However, current uses of FPGAs on SN come up with pure hardware nodes that have limited SW capability and thus are unsuitable for EC. EC requires nodes with (1) rich SW features, (2) flexible middleware for nodes communication and interaction and (3) real-time data processing capability. Hybrid FPGA devices consisting of an FPGA coupled with embedded processor are therefore good candidates. The processor greatly simplifies the implementation of an IP-based software stack and associated web services as well as provides enough performance for an edge-centric middleware. The FPGA is dedicated to critical and real-time data processing tasks.

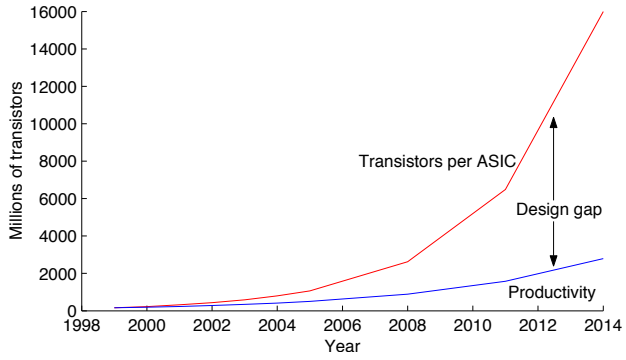


Figure 2.1: Hardware design productivity gap: number of transistors available on a chip *vs.* the ability for the transistors to be used efficiently in a design [int11, Sed06]

2.2 Hardware Design Background

2.2.1 Overview

Basically, a good system design methodology should consider following important criteria: (1) Time-to-market (design time), (2) Productivity (production outputs/inputs ratio), (3) Maintainability (repair/replace, flexible to change, easy to maintenance, etc.), (4) Extensibility (adding new capability) and (5) Reusability (reuse with less modification possible). A flexible and generic solution for software/hardware integration involves SW/HW co-design. Software design has shown an important evolution with Oriented Object Design and Design Pattern. These design methodologies have been well proven on real world projects. They are claimed to be the best methodologies to satisfy the presented criteria. In [LHKS91], authors have shown that object-oriented paradigm can improve productivity of software design by about 50% (1.5 times). Hardware design, however, alway remains a long and tedious process. According to More's Law, the complexity of hardware system in terms of logic transistors integrated on a chip increases about 58% per year. Nevertheless, the hardware design productivity increases around 21% per year [int11]. In Electronic Design Automation (EDA) community, this rate is known as *design productivity gap* (figure 2.1) which results in inflating design costs. Most current research efforts on hardware design are aimed at closing this gap by rising the abstraction level and increase IP (Intellectual Property) reuse.

2.2.2 Hardware Design Methodologies

The traditional and commonly used languages for hardware design is Hardware Description Languages (HDL), such as VHDL or Verilog. These languages allow to describe IP (Intellectual Property) cores at Register Transfer Level (RTL). For decades, the main concern of hardware designer is to create an efficient component (e.g. speed, area or power usage) using these languages. They focus on the creation and qualification of IP content for a specific application regardless of its further reusability in the application domain. This can cause a redundancy of similar IPs designed for different applications depending on different requirements. This design habit is called *content-based* design [DS04]. It is worth noting that reuse does not necessarily apply only to circuits (content), but can also be applied

to concepts and techniques. Hardware designers are using methodologies (concepts + techniques) that are several years behind software. Traditional HW design methods and HDLs are not equipped with modern design features –such as object oriented– and thus limit the reusability, extendability and maintainability of existing IPs.

The increased complexity of hardware system requires more productive methodologies. Currently, to gain productivity, modern hardware design seeks to rise the design abstraction level. Literally, there are mainly two directions of abstraction: *content-level abstraction* and *system-level abstraction*.

Content-level abstraction –also known as *High Level Synthesis (HLS)* [MS09]– allows to describe hardware IPs at an abstraction level higher than RTL using some traditional software languages like C, C++, SystemC, Matlab, etc. [Ren14, MVG+12]. These systems have a dedicated compiler that transforms an algorithm written in a target software language to a low-level RTL representation. This method improves the productivity of designers by providing automatic "correct-by-construction features" (via the compiler) and "separating correctness design concern from timing design concern" [PBMB16]. HLS methods are currently supported by FPGA vendors (Intel Altera, Xilinx, etc.) and commercial tools. They can be considered as future methodologies for *IP content-design*.

The *system-level abstraction* approaches, on the other hand, improve productivity by considering the abstraction at the system level in order to automatize the design process and to maximize the reusability of IPs. These approaches aim at a more *integration-based* solution for hardware components reuse. They encourage new hardware design habits [DS04, ŠD13]: (1) design-for-reuse rather than design-for-use (reusability), (2) designing for a generic problem rather than for a specific application (generalization), (3) customizing existing IPs for a problem rather than designing it from scratch (extensibility). Currently trending system-level abstraction methodologies rely on *meta-modeling* techniques using *Model Driven Engineering* (MDE) [Dam06]. The basic motivation is to model the hardware system at a higher level of abstraction and allows to describe hardware components (IPs) and their conceptual relationships at multi levels of abstraction. Therefore, designers need lesser focus on technological details and may benefit from high-level models for modern design techniques (i.e. object-oriented). This improves the productivity by increasing the modularity of the system (efficient reuse) and automating the design process (low-level code generation).

2.2.3 Discussion

To improve productivity, two abstraction methodologies –*content-level abstraction* vs. *system-level abstraction*– rely on two different strategies. One focuses on rising the abstraction of IP description and allows designer to work at algorithmic level rather than at RTL level. The other methodology, on the other hand, emphasizes the modularity of the hardware system and encourage the reusability and extensibility of hardware components (IPs). Although HLS approaches allow designers to describes hardware IP using decent software languages, they remain alway *content-based* design approaches and thus suffer the same reusability limit of HDL languages. Most research in this area focuses on scientific computing applications; it is not certain that HLSs is appropriate for system design. Approaches relied on MDE address to this problem by modelling the hardware system at a higher level abstraction. They are powerful to describe hardware components structure and relationships. However, it is not always obvious to express detailed behaviour of each component using these approaches. These issues

will be detailed in the following sections which will focus on the different *system level design* techniques using MDE.

2.3 Meta-modeling for System-level Hardware Design Using MDE

System-level design relied on MDE is becoming widespread since late 1990's [MC99, DMŠ03, BSP05, LBMD08]. Such design methodology has proven to be beneficial to deal with the increasing complexity of digital systems. Proposed solutions are diverse, some approaches define their own formal modelling semantic [BH98, WHMT08, IEE14] while others rely on well-known modelling standard like UML and Object Oriented Design [LW16a, EBZ+12, DS04]. These approches offer a high-level abstraction specification, APIs and tools for hardware system modelling. Model-to-HDL is available with or without restriction for supporting low-level synthesis on actual hardware. In this section, we first discuss about MDE in general, then consider three main categories of MDE methodologies for hardware system design: *Component-based*, *Platform-based* and *Object-oriented based* approaches.

2.3.1 Model-Driven Engineering

Before presenting different *system-level design* approaches for hardware design, we introduce –in this section– all methodological aspects related to *meta-modeling* with MDE. Model-Driven Engineering [RA12, VDMV14] uses abstraction to bridge the cognitive gap between the problem space and the solution space in a system. *Models* are the keys principle to describe the system at multi-levels of abstraction in order to close this gap. They are defined by formalism *Meta-models*. Models processing is mainly performed via *Model Transformations* which is considered as the "heart and soul of model-driven software and system development" [SK03]. Transformations can be used for code-generation, or for models mapping in the same or multiple levels of abstraction.

2.3.1.1 Model

The fundamental of MDE is model. A model is an abstraction representation of a real system or an environment. Models are based on generic concepts (or specific set of concerns) and their relations for system description and specification. MDE considers model as data that we can "query, analyze, report on, validate, simulate and transform in other useful formats" [OMG14]. More precisely, according to [Küh05], the model concept can be defined based on three main criteria: (1) *Mapping*: the model is related to the system by an explicit or implicit mapping between the reality and the concepts forming it. (2) *Reduction*: the model abstracts the system by representing only core properties that characterize that system and ignoring all unnecessary details. (3) *Pragmatic*: "A model need to be usable in place of the original with respect to some purpose".

2.3.1.2 Meta-model

A model represents an abstract view of a real system by using an abstract syntax defined by a *meta-model*. A meta-model describes precisely (1) the generic concepts and their relationships (used by models), (2) the structuring rules that constrain these concepts and (3) the combinations of the concepts

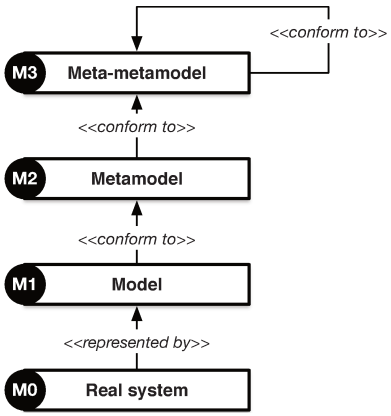


Figure 2.2: Four layers meta-modelling framework

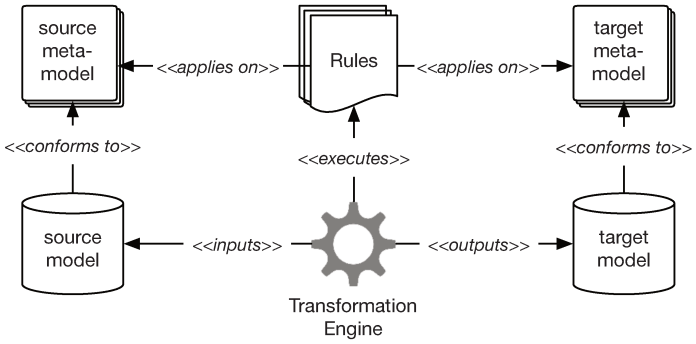


Figure 2.3: *Model transformations* allow to remodel an input model by executing transformation *rules* using a transformation *engine*.

in respecting to the structuring rules. In brief, a model conforms to its reference meta-model or, in other words, a model is written by a "language" defined by its meta-model. This relation is called *conformance* [Béz05].

4-level Metamodeling Framework

Figure 2.2 shows the four-level meta-modelling framework introduced by OMG in the Meta Object Facility (MOF) specification [Béz05, OMG15]. Level M0 represents the real system that need to be modelled by level M1 – the first abstraction level. The syntax of M1 (model), as presented, is define by a *meta-model* –at level M2. The meta-model –after all– is also model of model. Therefore, it need to be described by a meta-model at a higher abstraction level. This meta-model is called *meta-metamodel*, level M3. To avoid an infinite model tour, the meta-metamodel is claimed to be reflective. That is it can conform to itself, or in other words, it has the ability of self-definition.

As a part of our work, we use this modeling architecture to develop our object-oriented meta-model of FPGA circuits. This will be detailed in the following chapter.

2.3.1.3 Model Transformations

Model transformations are key operations of MDE. They allow to transform a model conforming to a source meta-model at an abstraction level into another model conforming to a target meta-model at the same or different abstraction level. Figure 2.3 shows the basic principle. Transformations are performed by applying *transformations rules* to the input model using a *transformation engine*. *Rules* are software artefacts that implement basic transformation steps. A transformation engine executes a collection of rules on a source model and derive the target model.

Model transformations can be classified differently according to different criteria. However, – in general– all transformations can be categorized into two classes: *Exogenous* and *Endogenous* [MVG06], as shown in figure 2.4.

A transformation is endogenous when the source model and the target model conform to a same meta-model. This operation can be an optimisation, a restructuring of model or a combination of individual models to form a complete model. A sub-category of this type of transformation is defined as *model composition* or *model weaving*. This operation allows to combine several models in to a single one and presents the links between model elements. It is used in applications that require traceability, model comparison or model annotation [RA12]. This model weaving concept is similar to the aspect weaving concept presented in *aspect-oriented programming* [Jéz08].

On the opposite side, a transformation exogenous is a transformation that derives an output model conforming to a different meta-model from the source meta-model. This category can be separated into two sub-categories: *horizontal* and *vertical* based on the abstraction levels. Horizontal transformations mean that the two meta-models (source and target) share the same abstraction level. This operation performs the migration of models between different aspects or domains within a system at the same level of abstraction. Vertical transformations, on the other hand, allow passing models between different abstraction levels. This can be a refinement (top-down) or a generalization (bottom-up) of model [FB01]. *Model-to-text* or *Text-to-model* are special kinds of vertical transformation. Model-to-text is used mostly for generating source code from low-level model and text-to-model defines the reverse engineering to construct model from source code.

Transformation rules allow mapping between concepts and relationships described by source meta-model to the corresponding concepts and relationships defined by target meta-model. These rules can be implemented by using software functions and procedures or by using a dedicated language. this language can be a *declarative* language (e.g. ATL [JK06], QVT [OMG16], etc.), which describes what will be produced by the rule, or an *imperative* language (e.g. Xtend and Xpand [Kla07], etc.) that specifies how the rule is executed.

2.3.2 Component-based approaches

In software engineering, Component-based design is a process that emphasizes the design and construction of SW systems using reusable software components [Fab07, Spa13]. It shifts the design perspective from programming SW to composing SW systems by mean of separation of concerns, in respect of the wide-ranging functionality of a given SW system. When it comes to HW design, Component-based design methodologies usually handle "the problems of representation, retrieval and

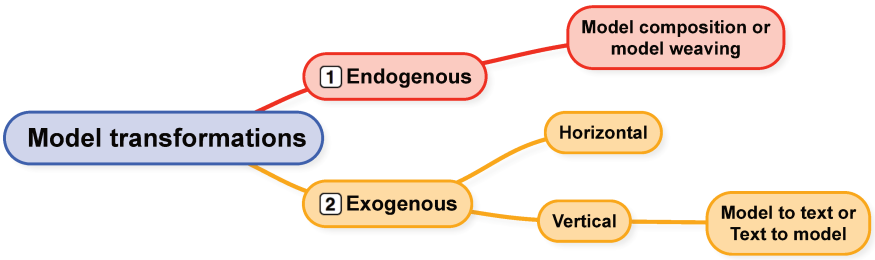


Figure 2.4: Classification of model transformations.

reuse of hardware components for IP libraries, IP providers and IP users" [Dam06]. These methods enhance IPs reused by offering a more flexible customization of IPs in response to the different application contexts. This is achieved by using design space exploration and parameterization methods. System modularization is the main concern of these approaches. They consider the hardware system as a network of connected components, then each component and its connections can be expressed, abstracted and parameterized using a high-level abstraction description meta-model. The proposed solutions are often language-centric. Unlike RTL HDL –which is emphasized on IP behaviour description– the proposed languages are component-centric. They are powerful for IP integration by offering a high-level abstraction concepts of components' interface (e.g. channel, synchronised processes and data flow, FSMD , etc.). Low level complex interface – in HDL– can be automatically generated from these concepts.

In [Zhu01], authors present MetaRTL, a RTL abstraction which extends the FSMD (Finite State Machine with Datapath) with a rich set of constructs and abstract type system (similarly to software) for IP design and integration. Synchronous hardware components are expressed by using a "class-like" construct which consists of fields and methods. This construct is not exactly class, it is only used for encapsulating a piece of synchronous hardware with ports, signals and behaviour. However, it does not offer the object-oriented characteristics as traditional class does (inheritance, polymorphism, etc.). Components interactions are simply performed by method call. A FSMD will be generated automatically if necessary for the interaction using a dedicated compiler. This approach resolves two main problems: (1) reduce design error-prone problems using abstract type system and memory abstraction and (2) improve IP reuse by automatically handling handshaking protocol between components (using FSMD).

In industrial scope, reuse of existing IPs is supported by a well-known standard called IP-XACT [IEE14]. It is firstly developed by SPIRIT Consortium –before merging with Accelera– and is lately standardized by IEEE. IP-XACT defines a set of unified structures (meta-mode) based on XML that enable the meta-specification (decoration) of a design, components, interfaces, and interconnection of components expressed in RTL form (e.g. HDL). The goal is to provide to IP designers/providers with the possibility to package and publish theirs IPs and to IP users with the ability to successfully use third party IPs in theirs system. IP-XACT can be used for both manual and automatic system design methodologies. It increases productivity by enhancing automation for IP selection, configuration (parameterization) and integration through plugin/EDA tools. IP-XACT does not come with tools for model transformation

and code generation, it is the responsibility of user to implement plugins, EDA tools conforming to the standard.

The work on [WHMT08] introduces another approach of using MDE for component-based hardware system-level design. This approach relies on COLA –COmponent LAnguage– [HKT+07], a synchronous dataflow language and tool. COLA has its own well-defined formal modelling syntax and semantic. The COLA tool allows the system high level specification as well as the generation of VHDL description, both for structures and behaviours. In general, a hardware system is modeled as a network of *units*, which can be composed hierarchically. Interactions between units/components are performed via *channels*. The network is controlled by an *automata*, which is actually a finit state machine diagram. This approach improves the productivity by raising the design abstraction. Beside, it handles the component integration problem by using abstract synchronous data flow models. GUI editor (e.g. for FSM diagram) reduces design complexity and avoid error-prone design tasks. COLA has also polymorphism components and library system for encouraging model reuse.

Discussion

The COLA and MetaRTL design approaches follow a "top-down" design method. The hardware components and theirs connections are specified at a high level abstraction using models. The low-level code (e.g. HDL) –both behavioural and structural– can be generated from these models following a predefined syntax and semantic. Therefore, the backward support to reuse existing HDL components is not possible. The IP-XACT approach, on the other hand, uses a "bottom-up" method. It is dedicated only to meta-specify existing HDL components. Since it enhances the automatic configuration and integration of IPs and thus improves reuse, it does not reduce the complexity of IP design. Unlike others approaches, where IP design and integration can be specified using a single high-level abstraction meta-model, in IP-XACT, designers need to handle separately the design task (in HDL) and the packaging task (in XML) of the IP. Changes made on one task can cause the modification on other task. This is error-prone and unnecessary.

Component-based approaches focus on the modularity of the hardware system. They resolve the integration problems using high level abstraction concepts, design space exploration and code generation. Therefore, these approaches improve reuse compared to traditional approaches. However, for the problem domain abstraction, despite efforts of merging OOD in component-based design [SDTF12, SDTF13] (in SW design), component-based approaches are not alway as flexible as Object Oriented approaches (abstract, inheritance, polymorphism, etc.) in term of reusability and adaptability. Furthermore, these approaches often rely on their own formal modelling semantic which, sometime, is not standard (apart from IP-XACT), thus, hindering its widely adoption on other system.

2.3.3 Platform-based approaches

Platform-based design* [SVM01, PBSV+04, Sed06] is a methodology for creating a highly integrated design. It improves productivity chiefly through extensive and planned IP reuse. A platform represents an abstract view of an application domain. Its architecture is based on a fix set of generic components

*In MDE, the term "platform model" is also know as "architectural model"

that can be parameterized at certain degree. Such platform allows to design various hardware systems in a given domain (e.g. video/image processing) and thus attain some generalization. Applications development is based on the composition and parameterization of generic components. Platform can be built on top of each other where the lower level design is abstracted away [SV02].

Platform-based design defines a fixed architectural model for an application domain based on two key principles [KNRSV00]: *interface standardization* and *Orthogonalisation of concerns*. Interface standardization means abstract communications between modular circuit components. This abstraction is mapped to physical interconnection by communication protocols which are defined by interconnection type (e.g. point-to-point, shared bus, network, etc.) and topology. Orthogonalisation of concerns means isolation of different design aspects so that they can be independently implemented, optimized, and explored. This isolation could be the separation of communication from computation or of function from architecture. These two principles promote design reuse by allowing the independent evolution of hardware blocks with respect to the system architecture. A component can be reused on different applications as long as it responds to all requirement to the architectural constraints.

Application design can be performed either in bottom-up or top-down manner. In the fist case, the system has a library of IPs, cores, or virtual components implementing high complexity functions (e.g. image processing). The application is built by selecting the component designs from library and connecting them together. In the other case, the library consists of only basic generic components (e.g. multipliers, vector processing etc.). they are automatically mapped together as the result of a top-down design process. Both methods must respect the pre-defined system architecture to ensure "generalization" of the platform.

Platform-based design is basically used for system-on-chip design [CF16]. For example, in computer vision domain, the work in [Sed06] proposes a platform –Sonic-on-Chip– with architecture and module libraries dedicated for reconfigurable video image processing. The HW application can be construct at run-time using dynamic reconfiguration. [Nga11] provides a network-on-chip for processing multi-image sensors. This network makes it possible to dynamically manage different flow in parallel by automatically adapting the data path between the computing units in order to efficiently execute different applications. Or, most recently, [KTO16] has reported the benefit of platform-based design for real-time image and video processing. Platform-base design can also be used for communication-based SoC architecture [Mar16, CF16] such as bus-based [OWTK10] or network-on-chip [Nga11] systems; or for multi-interconnect processors [Pom16, SF16], etc.

Discussion

Platform-based design shows interesting use on domain specific applications. By emphasizing on interface standardization and orthogonalisation of concerns, it imposes constrains on system architecture and thus allow efficient reusability and scalability (on large designs). Since it focuses on an application domain, platform-based design can be opted for specific system performance enhancement (e.g. image processing). Moreover, platform-based design is not mutually exclusive, that is, it can be used conjointly with other design methodologies such as HLS, component-based or object oriented based design for better productivity. The architectural model can also be applied on SW/HW co-design that is helpful for automatic SW/HW communication.

2.3.4 UML and Object Oriented based Approaches

In the context of hardware system-level design, the term "Object-oriented" (OO) means the application of Object-oriented Design (OOD) principles on hardware design using MDE rather than the use of object oriented languages (OOLs) for hardware description. There are work that did use OOLs for hardware description such as [GL95] (Ruby), [BH98, HBH+99] (Java,JHDL), [Dec04] (Python), etc. These approaches have dedicated API that allows to describe hardware design using an OOL. They attain some productivity improvement by abstracting data type and constructs and by avoiding the syntax verbose problem of traditional HDL. Nevertheless, despite the use of an OOL, the application of OOD principles (abstraction, inheritance, polymorphism, etc.) on hardware design itself is missing or limited, making these approaches unsuitable for system-level design.

OO-based system-level design approaches share some similarities with the component based approaches such as system modularization, auto code generation, etc. However, they use a more flexible abstraction models for problem domain abstraction that support better IP reusability and adaptability. Moreover, like on software domain, the use of OOD on hardware domain also improves the documentation for the further reuse and system maintenance.

Most recent OO-based approaches rely on the well-known standard modeling language UML [OU15]. UML, developed by the Object Management Group, is an open standard language for specifying, visualizing, constructing and documenting the software systems. However, it can also be used to model other non software system. UML has a rich set of diagrams that allow to specify –at a hight level specification– the structure (e.g. class diagram) as well as the behaviour (e.g. state or sequence diagram) of a system. Several UML profiles have been proposed for hardware system modelling such as UML for SoC [VMD08], UML of System C [RSRB05] or the standard MARTE [OU11]. MARTE, stand for *Modelling and Analysis of Real-time and Embedded Systems*, is an UML profile for model driven development of real time and embedded system (for both functional and non-functional aspects). It is used in most recent proposed approaches for hardware system-level design.

Most proposed approaches allow the mapping between UML diagrams and HDL concepts. Several works model the hardware system behaviour using UML behavioural diagram with different mapping models. MODEASY [WAU+08] introduces a method and modelling tool for transforming a subset of UML state diagram elements into synthesizeable VHDL description. It allows to specify system behaviour in form of FSM. The work in [DA13] proposes method for mapping UML state diagram to an intermediate model based on hierarchical configurable Petri net. Then, this intermediate model can be converted to HDL such as VHDL or Verilog. Another mapping possible is shown in [BAS14], this method allows transformation from a state diagram using temporal Hierarchical Concurrent Finite State Machine (HCFSM) model, into Verilog hardware specification, the generated HDL can be simulated or synthesized on hardware. A part from state diagram, there are some other works that use a sequence diagram instead, such as [EFQ15, LW16b]. The proposed approach in [EFQ15] uses UML/MARTE to model the hardware system, it starts from a a sequence diagram with timing constraints, then automatically generates its implementation in both System C or VHDL for verification.

Using UML behavioural diagram allows to specify the structure and functionality of interacting components using a high-level abstract model and hence reduces the design complexity. However, since these kind of diagrams focus only on behavioural description, they are not appropriate for structural

specification, which is important in system-level design. Furthermore, these approaches do not use the main principle of OOD (e.g. inheritance, polymorphism, etc.), and thus limit the reusability and adaptability of the system. Other modelling approaches address this problem by using UML structural diagrams.

GASPARD [GLBP+11, QMD08, EBZ+12] is a MARTE compliant SoC design tool and environment that allows to obtain executable system (such as hardware system in VHDL) from a high level MARTE specification. It relies heavily on the concepts presented in the *Hardware Resource Model* package. Such tool focus on the system modularization (using component diagrams) and components mapping, thus allows efficient system structural specification. The system allows to generate structural skeleton VHDL codes which consist of entity and components mapping. The component behaviour must rely on existing IP or be described manually (using HDL).

The work in [DMŠ03, DS04] brings the principles of OOD and design pattern into hardware design. It proposes a framework for OOD concepts, domain specific concepts and meta programming on hardware design. The system focus on (1) the structuring, the enscaptsulation, and reuse of HW designs at a highest level of abstraction; (2) using OOD techniques not only on hardware system modelling but also on hardware design processes. The work uses class diagram for system specification and is able to map between UML class diagram and structural VHDL abstractions (skeleton code). The component behaviour is manually described using HDL or meta-programming. Some software design patterns such as *composite*, *decorator*, *adapter* have been adapted on hardware design for efficient IP reuse and customizing.

Most recently, AMoDE-RT [MWP+10, LW16a] is a MDE approach compliant with MARTE that target the FPGA-based embedded realtime system design. UML models are intensively used to generate system implementation. It provides a script-based tool called GenERTiCA for mapping between UML model element and VHDL structures. The system proposes a complete design solution by supporting to describe both structure (using class diagram) and behaviour (using sequence diagram) of the hardware system. Some basic OOD design principles (without design pattern) are allowed, such as encapsulation, inheritance, association.

Apart from UML, there exist also other OO approaches based on OO HDL such as System C. [MGSPF11] introduces a case study of using Aspect Oriented Programming and OOD in hardware design using System C. The work focuses on increasing components reuse by decreasing components coupling. The proposed approach supports the use of OO concept (inheritance, interface), design pattern, and meta-programing on the design process.

Discussion

OOD techniques promote productivity by describing the system in an abstract and implementation-independent way. They raise significantly the abstraction level and encourage system reusability and adaptability. This has been well proven on software domain. From the conception perspective, hardware systems share some similarities with OOD concepts (components vs. classes, component communication vs object communication, etc.), and thus can be expressed using OOD.

UML shows an important application in high-level system design. It allows to combine the OO concepts for structural design (component, class diagrams) with the behavioural models (state, se-

quence diagrams, etc.). Some works focus on the use of UML for modelling system functionality that can produce behavioural HDL code from graphical diagrams. However, they lose the support of OOD techniques in the design process. Other works emphasize on system structural specification. OOD techniques is the primary concerns of these approaches. They have shown that, the use of OO concepts and design pattern in hardware can solve the design reuse problem using abstract concepts. Nevertheless, they are hard to specify the system behaviour which needs to be described manually or via meta-programming. Only the work on [LW16a] proposes a complete solution for both system structural and behavioural.

Because of the natural difference between UML high level graphical aspects and low level hardware concepts, it is not obvious to use all possible diagrams (structural and behavioural) elements to express hardware concepts. Most proposed solutions use only a subset elements of a target diagram to model hardware system. This constrains the design process to a subset hardware elements and patterns, which sometime, limit the design flexibility and the scalability of system. We argue that there is the need of an *intermediate meta-model* to fill in this gap. A meta-model that is familiar with high-level OOD design principles –and therefore UML compliant– while being flexible enough to express low level hardware behaviour and structure.

2.3.5 Summary

To deal with the increasing complexity of digital systems, MDE gains advantage over traditional approach by: (1) reducing system complexity via high-level abstract model, (1) encouraging design reuse and adaptation by mean of separation of concerns, generalization and generation (to HDL), and (3) aiming at integrating different existing design methods and tools into a unique high-level design environment and flow. Table 2.1 shows a review of presented approaches using MDE, based on the *reusability*, *adaptability* and *extensibility* of a HW design. Platform-based approaches particularly address a specific application domain. They rely on common architectures based on principle components fixed within a certain degree of parameterization. Such architectures supports a variety of application in a given domain, thus attain some generalization. Component-based and OOD approaches are more generic. Component-based approaches are service-oriented and best at system functional abstractions. They are powerful for IP integration by using a high-level abstraction concepts of component communication protocol. OOD approaches, on the other hand, are identity-oriented and are best at abstracting the problem domain of a system. They offer a more flexible model to describe, reuse, adapt and extend hardware IPs. OOD can be integrated in platform-based design models in order to add more abstraction to the architectural model, thus enhances the components reuse and integration in an application domain.

Solution	Design methodolology	Modeling syntax/semantic	Abstraction level	Design adaptability	Inheritance (Extensibility)	Legacy HDL reuse
MetaRTL	Component-based	Language-centric meta-model	Functional, protocol level	no	no	no
IP-XACT	Component-based	XML	Protocol level	Parameterization	no	yes
COLA	Component-based	Language-centric meta-model	functional, protocol level	Polymorphism	no	no
[Sed06], [Nga11], [KTO16], [OWTK10], [Pom16]	Platform-based	Architectural model	Application domain, interface standardisation	Parameterization	no	Fixed set of generic components
MODEASY, [DA13], [BAS14]	Object-oriented	UML	Functional (state diagram)	no	no	no
[EFQ15], [LW16b]	Object-Oriented	UML	Functional (sequence dia.)	no	no	no
GASPARD	Object-oriented	UML	Structural (component dia.)	Parameterization	no	no
[DMŠ03], [DS04]	Object-oriented	UML	Structural (class diagram)	Design pattern	yes	no
AMoDE-RT	Object-oriented	UML	Structural (class dia.) & functional (sequence dia.)	abstract methods	yes	no

Table 2.1: Synthetic table of presented HW system-level design approaches using MDE

A common problem of current MDE design approaches is that they only allow one-way mapping between high level concepts and low level hardware elements (HDL). The inverse process that efficiently reuses existing HDL designs in MDE environment is not considered enough, apart from IP-XACT which is only intended for regular IP reuse. By reuse, we mean the use of HDL designs, not as a low-level static elements but as high level active design elements. It is important to note that most currently existing legacy IP library are designed using traditional HDL. Allowing reuse of regular HDL designs in MDE environment should be considered as an important factor of a productive design system.

As shown in table 2.1, none of presented approaches satisfy all the criteria (reusability, adaptability, extensibility and legacy HDL reuse). Our objective is to propose a design approach aiming at fulfilling all these criteria. We argue that OOD is suitable for our purpose since it naturally promote the *integration-based* design concepts (presented in section 2.2.2): (1) design for reuse (abstraction), (2) design for generic problem (problem domain abstraction), (3) design for easy adaptability and extensibility (design pattern, inheritance). However, we do not try to directly map between high level OOD concepts to low-level hardware structures. We aim at filling this gap using an intermediate meta-model that bring OOD principles to HW design.

2.4 Software/Hardware Co-design

SW/HW integration is another factor that has a significant impact on the overall productivity of the embedded system design process, mostly on complex systems which requires simultaneous development of software and hardware. SW/HW interaction problem remains always problematic. It varies from project to project due to the change of requirements and takes an important amount of development time while has less contribution on the overall system functionality. Such tedious, unnecessary and error-prone task, need to be taken in to account in SW/HW system level design.

SW/HW co-design [Tei12, JDM+07] addresses this problem by abstracting the SW/HW system-level design process. It meets the design objectives by exploiting the synergism of hardware and software through their concurrent design. This is achieved using various methodologies for *design space exploration*,*generalization* and *generation*. SW/HW co-design provides a heterogeneous design environment that includes descriptions of SW/HW, and communication modules. Literately, proposed solutions for SW/HW co-design can be classed into two categories : *Early binding* and *late binding* approaches.

2.4.1 Early Binding Approaches

Early binding means that the SW/HW separation of the system is performed at the early stage of a top-down design process. In complex SW/HW system, this reduces the design complexity by decomposing the system into a hierarchy of manageable and coordinated sub-systems. Figure 2.5 shows the basic steps of the design process. At the beginning, the designer performs a system analysis and identification of tasks, from here, the system is separated into two parts (SW and HW). Both parts are then developed simultaneously and separately. Usually, the hardware part is modelled in a high-level abstract form (e.g. MDE, HLS, etc.), that afterward promotes the SW/HW integration. It may be constrained to some architectural requirements if needed. The mapping stage is an automatic or

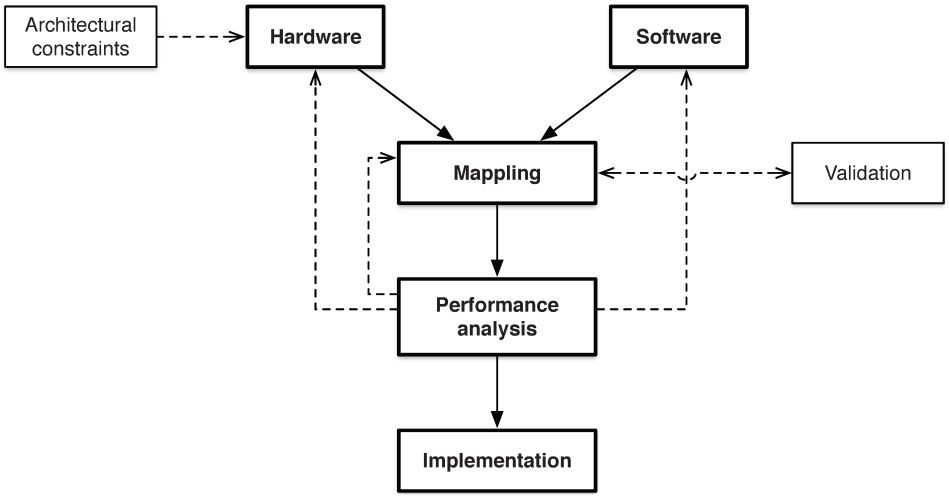


Figure 2.5: *Early binding*: Both SW/HW parts are designed simultaneously and separately. An automatic mapping process is needed to interfacing both parts

semi-automatic operation that performs inter-tasks for SW/HW interfacing. The mapping mechanism should be flexible enough to respond to different kinds of project or to different underlying physical HW/SW links. This mechanism often relies on an automatic code generation process for the glue logic between SW and HW.

Most platform-based design approaches –as presented in section 2.3.3– use this design method for SW/HW co-development. These approaches apply architectural constraints to the entire SW/HW system, thus increase modularity of the system, both on SW and HW. The communication interface is standardized so that different applications can easily fit to it (as long as they respect to the proposed constraints). The work in [LSS13a, LSS13b, LSS14] presents Unity, a communication-based platform dedicated to integrate FPGAs to high-level robotic software system (e.g. ROS). It focuses on the interconnection architecture between FPGA based devices and a standard PC via a common physical interface (memory bus, USB, serial, ethernet, etc.). On the hardware side, it defines a unified stateful message-based IO interface on top of an address/data bus architecture for all user circuits. This interface architecture supports a variety of physical interface technologies. On the software side, it allows software to communicate with user logic by following a simple message-based protocol that give access to the shared memory model of the underlying hardware. However, the hardware design part of the platform always relies on a traditional design process (using HDL). Reuse of existing IPs needs to manually write VHDL wrapper for every each added core. The platform improves partially the design productivity of SW/HW system via interface standardisation and semi-automatic code generation, but the problem of traditional hardware design always remains unsolved. This platform based design approach could be enhanced by combining it with an high-level design method such as OOD or UML which enables better reusability and adaptability for HW design. Some of UML-based approaches (presented in previous section), such as GASPARD [GLBP+11] allow this combination for a complete high-level SW/HW system co-design.

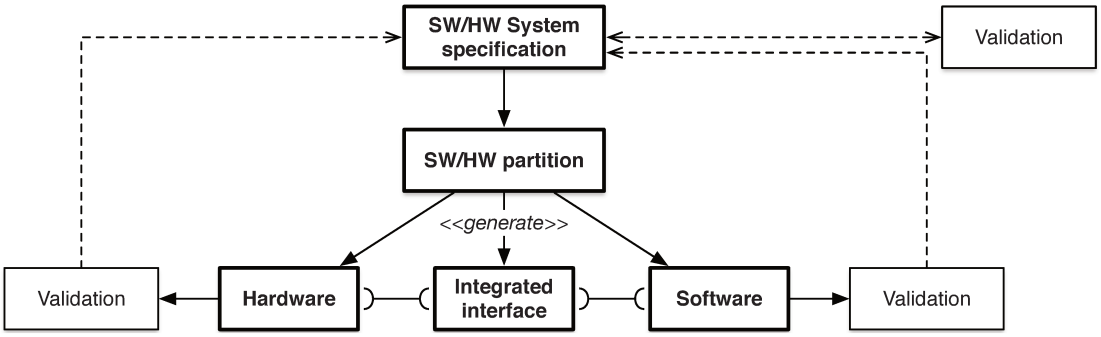


Figure 2.6: *Late binding*: Both SW/HW parts are specified by a unique model (language-based or MDE-based). The system handles automatically the partition and the inter-task for interfacing at the high-level-synthesis phase.

2.4.2 Late Binding Approach

In contrast to *Early binding* approaches, in *late binding*, the decision for SW/HW allocation are deferred as late as possible within the design process as shown in figure 2.6. At the design stage, the system is specified in an abstract form using a unique high-level model, without committing any task to either software or hardware. The system can be verified using a series of simulation or formal verification before passing to the partition stage. This latter will perform the automatic partition and implementation of tasks in SW and HW. The inter-tasks for interfacing and communicating are also handled in this phase. The generated SW/HW parts can be finally compiled/sythesized using a traditional design flow. Basically, there are two directions for this *late binding* design process: *language-based* and *model-based*.

Language-based approaches often rely on HLS [Ren14, MVG+12] as a development tool to offers the ability of using a same programming language for both software and hardware design. The system is partitioned into SW and HW tasks at the compile-time of the language using a dedicated compiler. Some HLS approaches, such as Molen [PBV07] or LegUp [CCA+13, FCC+14], have built-in support for co-design. The compiler is able to compile one (or more) selected C function(s) into hardware accelerators (FPGAs) while the remainder of the application is compiled to run on a processor. The communication between the processor and the accelerator is automatically handled at the instruction level. Often, these compilers are architecture dependent, which target to some specific kinds of hybrid SoC (FPGA coupled with processor). the LegUp compiler targets the Tiger MIPS processor, or a hard ARM Cortex-A9 while the Molen compiler is used for PowerPC General Purpose Processor. The SW/HW communication interface limits to a co-processor/processor interface.

Programming languages often miss formal semantics for specifying hardware nonfunctional properties such as timing or cost [Tei12]. In the case that these properties play important roles in system, *model-based approaches* is more preferable. As presented in section 2.3.4, a systems can be represented using high-level formal models (FSMs, Petri nets, data-flow model, etc.). These models allows to describe not only functional but also nonfunctional properties (e.g. UML/MARTE) of a system. They are architecture independent and can present both HW and SW. Model transformations can be used to provide implementation of tasks in either SW or HW depending on different mapping rules. *Edward et al.* [ELLSV01] explores various use of formal computational models such as Discrete Event, Commu-

nication Finite State Machines (FSM), Synchronous/Reactive and Data-flow Process Networks Models in SW/HW co-design. These models include: formal specifications, set of system and non functional properties and design constraints. The functional specification fully characterizes the system while satisfying the set of properties. In [CT05], authors propose a co-design approach based on the transformation of different UML diagrams into SW or HW tasks. The work introduces MODCO, a tools that help generate HDL code from UML state diagram. They state that the tool is the first step to bridge the gap between hardware and software design.

2.4.3 Discussion

SW/HW co-design is an important step for rapid prototype of complex embedded systems. It aims at an heterogenous architecture and environment (SW, HW and communication modules) for system design. The goal is to optimize the design constraints such as cost, performance and power as much as possible while reducing the time-to-market of the system. Trade off are often made between these requirements to achieve a reasonable design productivity[†].

Late binding co-design approaches often focuses on language-based design method. Despite the advantages of using unified language for specifying both SW and HW at a algorithm level, these approaches have some limits due to the nature of the language. Firstly, this latter usually invokes the use of a compiler which is architecture specific. Changing the underlying architecture (processor+ FPGA) leads to an important overhead to develop a new compiler that supports the new hybrid platform. This is not always faisaible. Secondly, the language emphasize on algorithm description, which is content-based and does not inherently encourage design reuse, as discussed in section 2.2.2.

A combination of *Early binding* co-design, with platform-based design and high-level system modelling (OOD, UML) offers a better system modularity (both on SW and HW), thus promotes the system reusability and adaptability. It allows also a better decoupling of the system with the underlying hardware platform. This increases the flexibility of the design environment with regard to different projects or different architectures (processor/FPGA and their interfacing).

2.5 Positioning our work

The main contribution of this thesis is to study the application of modern design principles –Object Oriented Design (OOD)– on hybrid hardware/software system design, in the context of IoT and edge-centric computing. The work focuses on three main design concerns:

The first design concern is *Hardware Design*. We introduce an object oriented meta-model that bring OOD principles to hardware system-level design. Our approach is based on the observation that: (1) most *content-based* design methodologies lack the ability to efficiently abstract the hardware at system level (coarse-grained level); (2) the majority of proposed *system-level* design methodologies are not fulfilled enough to specify the detail hardware behaviour (fine-grained, RTL level) and promote flexible IPs reusability, adaptability and extensibility; and (3) modern MDE-based approaches have difficulty to reuse and integrated existing legacy HDL designs in their environment. Our meta-model aims to

[†] the amount of design space exploration possible will be limited in complex system, thus the result is not alway optimal

fill the different gap between high-level design concepts (i.e. OOD concept) and low-level hardware concepts. It plays the role of an immediate meta-model that is familiar with most modern object oriented design methodologies, while being flexible enough to express any low-level hardware structures and have backward support to legacy HDL design reuse.

Second design concern is *Software/hardware co-design*. The application context is focused on the use of FPGA hybrid system for IoT edge-centric computing. The design environment here is limited to an application domain. Therefore, our solution on HW/SW design is based on a combination of *early binding* co-design approach with an object oriented platform-based system-level design approach. Object oriented design is used for both software and hardware design. The SW/HW system is highly modularized and reusable for different flavours of applications.

Last design concern covers the middleware design for IoT edge-computing. We provides an edge-centric middleware dedicated to the proposed hybrid system. This middleware eases the integration of FPGA edge-centric devices to the network. It supports distributed development and remote reconfiguration of nodes (both on SW at runtime and HW).

"Meta-design is much more difficult than design; it's easier to draw something than to explain how to draw it"

Donald Knuth, The Metafont Book

3

Promoting Object Oriented Principles on HW Design Using the OoRC Meta-model

Contents

This chapter presents OoRC (Objectification of Reconfigurable Circuits), a dedicated meta-model for integration-based HW design. It begins with an overview of the meta-model and its associated features. Section 3.2 enlightens about how the meta-model can be used for *content-based* design. Section 3.3 presents OoRCScript, a dedicated Domain Specific Language (DSL) for describing digital circuits using our meta-model. Hardware system-level design with object oriented technique will be presented in section 3.4. We talk about different kinds of model transformation in OoRC in Section 3.5. The simulation of circuit models is covered in section 3.6. Section 3.7 describes the interfacing of OoRC with external tools such as external simulator or synthesizer, etc. Finally, section 3.8 concludes the chapter.

3.1 Introduction

Content-based design methodologies (such as HDL or HLS) mainly focus on the creation and qualification of IP (Intellectual Property) content. They have very limited features that support efficient reusability, extendability and maintainability of IPs. Many *System-level design* methodologies use different design paradigm to address these problems. They aim at abstracting and automatizing the hardware design process at system level by mean of *generalization*, system *standardization*, *separation of concerns* and system *modularization*. That said, they are designed to handle the design task at a system perspective. However, when tearing down to the RTL level, these methodologies lack the ability to efficiently describe the content of each IP. Often, they rely on an external design methodology for IP-content design (e.g. MDE for system-level design while using HDL or HLS for content-based design). This ends up with an heterogenous design environment which is complex and error prone. Some methodologies support built-in IP content design but with limited structures or computational model (e.g. FSM, state diagram, sequence diagram, etc), thus are not flexible.

The OoRC meta-model offers a unique and homogenous environment for both hardware system level design and IP content description. It uses Object oriented paradigm for *System-level* design while having dedicated API and language for IP content design. It means to close the different gap between modern SW design methodology (OOD) and low level HW concepts and allows the two design directions can cohabit so that developers can benefit the advantages of both worlds in a sole environment.

3.1.1 OoRC in a Nutshell

3.1.1.1 Objectives

The OoRC meta-model aims at covering both *content-level* design and *system-level* design problem. At RTL level, it has API and dedicated DSL for modeling HW circuits as a graph of connected objects. Unlike most RTL approaches that rely on static meta-model for defining passive and rigid models of circuits, OoRC's models are able to evolve dynamically thank to the live objects. This feature simplifies and promotes the (run-time) (semi-)automatic models processing, such as: (1) structural refactoring of circuit models for optimization; (2) automatic injection of structures and behaviours to a circuit model depending on application context (e.g. debug sub-circuit); (3) automatic incremental construction of circuit model via GUI (CAD) tool or DSL; (4) definition of abstract structures from OoRC structures, providing correct-by-construction feature and automatic structure and code generation, etc.

At *system-level*, OOD techniques and design pattern can be directly applied on circuit models. This provides the abstraction for presenting a design solution of a problem in a implementation-independent way. The main philosophy here is that the subject of OOD is not a physically existing object (objects at RTL level), but the abstract concepts for solving a design problem. This separation of concerns allows better design modularity, reusability and adaptability.

In OoRC, when talking about design reuse, we do not only mean the reuse of models represented by our meta-model but also the reuse of traditional HDL models. The OoRC provides backward support to legacy VHDL IPs. That is, out meta-model is able to import and present VHDL IPs as regular circuit models which then can be used to perform model processing, model integration, etc. As stated, this feature is important since the majority existing IP libraries are in traditional *HDL* forms.

3.1.1.2 Features

The meta-model is equipped with dedicated libraries for structural and behavioural modelling of hardware structures (at RTL level as well as system level). It is implemented using Pharo Smalltalk, an object oriented language and environment suitable for system modelling. On top of these APIs, different features are available:

- OoRCScript is a DSL based on Smalltalk syntax. It allows to design FPGA circuits right inside the programming environment.
- OoRC supports OOD and *design pattern* for hardware system-level design. These techniques can also be used for model processing and automatic IPs integration.
- To support reuse of legacy VHDL, the system relies on a dedicated VHDL parser to import and transform a HDL model to OoRC model.
- Circuits models support both *in-vivo* (internal) simulation and *ex-vivo* external simulation (e.g. GHDL).
- Circuit models can be exported to VHDL for synthesis. OoRC has a toolchain that automatizes the circuit synthesis using vendor tools.

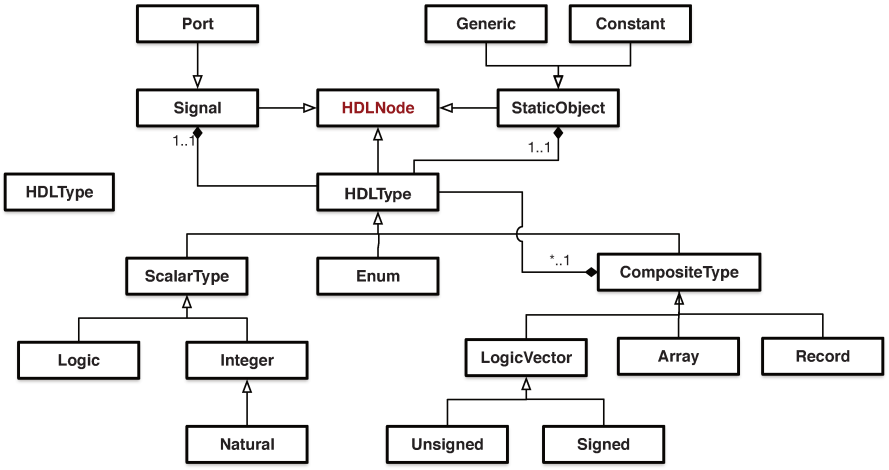


Figure 3.1: Simplified class diagram models signals and their associated data types. OoRC supports only synthesizable data type

The meta-model can also be used to build high-level tools (CAD,UML-based tools, etc.) for system design or for automatically processing models (code generation, automatic refactoring, etc.). In chapter 4, we will show how OoRC can be used to support efficient software/hardware co-design.

3.2 Fine-grained Modeling: FPGA Circuit at RTL Level

To describe FPGA circuits, our meta-model relies strongly on the well known hardware description language VHDL. The language comes with many extensions and can be used for hardware description or simulation purposes. Since our objective is to model circuits that can be synthesized on actual FPGAs, the meta-model is based on a subset of VHDL dedicated for the synthesizable system: the IEEE 1076.66 RTL synthesis standard [IEE04]. It can describe all basic VHDL structures in the standard*. Such structures are called meta-descriptions in our meta-model. A meta-description contains structural and behavioural informations about an element of a circuit (data, operation, control or other sub-circuit). Thereby, for each individual element, we know how it is organized, how it connects to other elements and how it is performed (executed) when activated. In this way, when all elements are connected together, we have a full structural and behavioural description of a circuit.

In OoRC, all meta-descriptions are subclasses (directly or indirectly) of a single class named *HDLNode*. These classes have methods and protocols that define the architecture and behaviour of an associated circuit element.

3.2.1 Circuit Signals as Data Objects

The state of a circuit at a particular point in time is represented by values of its ports and of its internal signals. The circuit-model, apart from the specification, contains also the data objects which refer to these signals. A Signal is an object of a data type that represents either wires, or output of a combina-

*Archive structures (package, library, etc.) are not directly supported. However, they can be indirectly described using object oriented technique

tional logic, or latches or registers. A Port is a Signal with additional information about its direction. In our meta model, a Port must be either input, output or bidirectional (inout).

As shown in figure 3.1, the meta-model supports basic synthesizable data type supported by most synthesis tools:

- Integer, Natural, Logic, LogicVector, Signed and Unsigned (in accordance with the *integer*, *natural*, *std_logic*, *std_logic_vector*, *signed* and *unsigned* types in VHDL): for basic arithmetic and logic operations.
- Record (collection of elements of different data types); and Array (collection of similar elements of any datatype) described using a composite pattern.
- Enumeration for enumerated and user defined type.

In digital circuits, each operation has a propagation delay, and thus the assignment of its outputs to signals needs also a delay to take effect. In OoRC, we model a Signal as an object with history. To maintain this time history, a signal holds two informations: (1) the current value of the signal (before the operation) and (2) the new value that will take effect after the propagation delay of the operation. Each time the signal is updated (i.e. once the propagation delay elapses), the new value will become current.

Note that, in OoRC, variables are modeled as an instance of a data type class. They hold one value at a time and have no history. Variables are used in processes, functions or procedures.

3.2.2 Circuit Structures Modeling

Figure 3.2 shows a simplified class diagram of circuit's meta-descriptions. The class *HDLDesignEntry* models a complete circuit with an interface (*HDLEntity*) and one or more architecture(s) (*HDLArchitecture*). An entity specifies the external interface of a circuit while an architecture describe its internal structure and behaviour. A circuit may have different architectures but only one is used at a time. A subclass of *HDLDesignEntry* –when created or changed– will be assigned automatically a signature number. Therefore, all instances of it (models) share the same signature. This signature number is synthesizable as a constant signal on actual hardware. It has nothing to do in circuit modelling but is useful to verify whether the circuit is deployed on actual hardware.

All subclasses of *HDLStructure* are meta-descriptions that describes circuit elements (i.e. sub-circuits). They allow to: (1) specify the structure of an element *and* (2) define the behaviour of that element in responding to the change of its inputs (signals). Circuit elements can be broadly classified in two domains: *combinational domain* and *sequential domain*. *A combinational circuit* has no internal memory (i.e. latches or fliplops) or state (i.e. closed feedback loop). Its outputs are defined as a function of inputs only. The same input value will always produce an identical output (when settled). In OoRC, all synthesizable VHDL concurrent structures (e.g. *conditional signal assignment, selected signal assignment, etc.*) are modeled as subclasses of *HDLConcurrent*. Note that, although *processes* are composed of sequential statements,they are defined as a concurrent structure. They are executed in parallel with other processes or other concurrent statements.

A sequential circuit, on the other hand, has an internal state and its output is a function of inputs as well as the internal state. Although sequential circuit can be described using concurrent statements

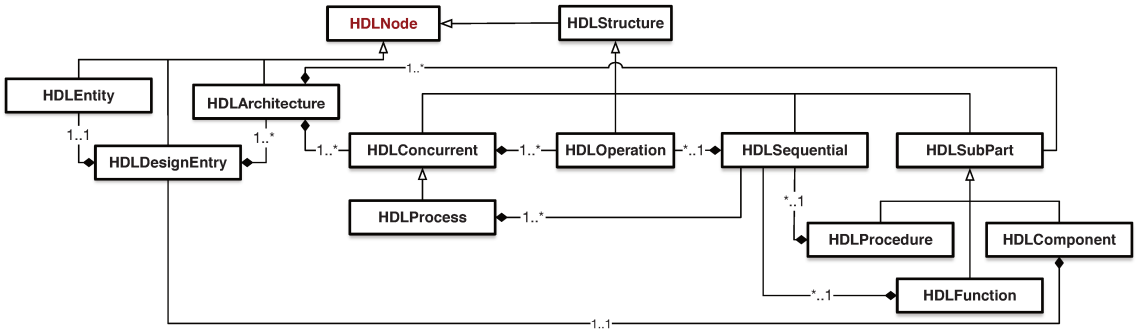


Figure 3.2: Classes define meta-descriptions of hardware structures. Theses meta-descriptions are based on synthesizable VHDL structures

(*HDLConcurrent*), in OoRC, only *HDLProcess* is used to specify sequential circuit. A process is composed of sequential statements. These statements are modeled as subclasses of *HDLSequential*. The meta-model supports following VHDL statements: *if statement, case statement, loop statement, signal assignment, variable assignment*. It supports also procedure call and the return statement from a function call.

Beside *HDLProcess*, *HDLProcedure* and *HDLFunction* can use sequential meta-descriptions to specify a procedure or a function. VHDL Procedure and function are modeled in OoRC for compatibility purpose when reusing VHDL legacy. A circuit model can use other circuit models as sub-circuits, this can be described using *HDLComponent*.

3.2.3 Discussion

OoRC captures the base principles of VHDL and allows to model circuits as a graphs of connected objects. A circuit model is able to evolve dynamically and thus is suitable for model transformation/processing. In addition, as a live object, it can capture not only the structure but also the behaviour of the circuit. This yields an executable model which can perform *in-vivo* circuit simulation without the need of an external simulator. The main objective is to bring the low-level hardware concepts to the high-level object oriented concepts and pave the way to fully use OOD principles on HW design. Abstraction can be easily added to these concepts in order to provide problem domain abstraction.

```
1  library ieee ;
2    use ieee.std_logic_1164.all ;
3    use ieee.numeric_std.all ;
4  entity SimpleFIR is -- External interface
5    port (
6    clk,reset:in std_logic;
7    start:in std_logic;
8    sample: in std_logic_vector(31 downto 0) ;
9    filtered: out std_logic_vector(31 downto 0));
10 end SimpleFIR;
11 architecture arch of SimpleFIR is -- Internal architecture
12   signal x_prev, x_prev_next, y_n, y_n_next:unsigned(31 downto 0);
13 begin
```

```
14    -- combinational logic
15    x_prev_next <= unsigned(sample) when start = '1' else x_prev;
16    y_n_next <= unsigned(sample) + x_prev when start = '1' else y_n;
17    filtered <= std_logic_vector(y_n);
18    -- sequential logic
19    process( clk )
20    begin
21      if reset = '1' then
22        x_prev <= (others=>'0');
23        y_n <= (others=>'0');
24      elsif rising_edge(clk) then
25        x_prev <= x_prev_next;
26        y_n <= y_n_next;
27      end if;
28    end process ;
29  end architecture ;
```

Listing 3.1: VHDL implementation of a simplest low pass FIR filter $y_n = x_n + x_{n-1}$

3.3 A Simplified DSL for HW Design

3.3.1 Overview of the DSL

OoRCScript is an embedded DSL in the Pharo environment and is based on OoRC API for circuit description. It is extensible, compiler-free and inherits all characteristics of an interpreted language as well as the dynamic environment of Pharo. With its compact and minimalist syntax, OoRCScript simplifies and abstracts the circuit specification in comparing to the API. While being able to express low-level hardware concepts (using the API), the language is naturally object oriented. Hence, it is familiar with most object oriented design principles. The OOD concepts can be used directly on OoRCScript for system level design. The same language is used for both HW content and system level design. Furthermore, OORCScript is not mutually exclusive, it can be used conjointly with other system-level design methodologies such as UML. Since the language is object oriented friendly, mapping UML diagrams to OoRCScript is more straightforward and easier than to a traditional HDL language (e.g. VHDL or Verilog).

3.3.2 OoRCScript Syntax

The meta-model has dedicated APIs for circuit model construction. This is helpful for automatic model processing or building high level tools. One can use these APIs to manually describe the circuit, but it is more preferable to use OoRCScript, a dedicated DSL for this purpose. This DSL allows to simply and directly describe the FPGA circuits. Figure 3.3 shows the example use of the DSL for describing the FIR circuit presented in listing 3.1[†]. A circuit model can be described by subclassing the class

[†]OoRCScript has been evolved since the last version presented in [LLF+15] by eliminating the need of a compiler and improving/simplifying the syntax

Figure 3.3: Description of the FIR filter in listing 3.1 using OoRCScript

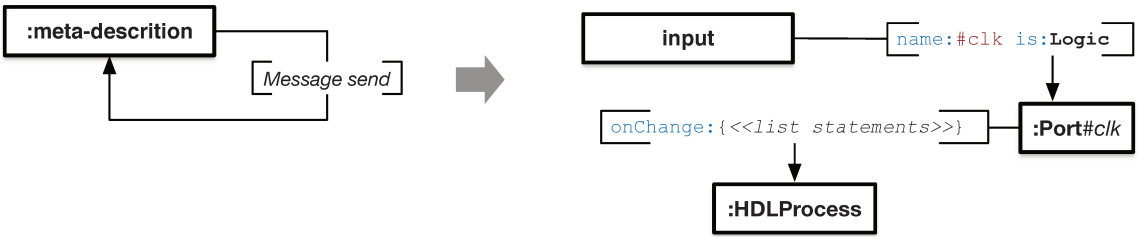


Figure 3.4: Basic principle of OoRCScript syntax: ”sending a message to a description will generate a new description”

HDLDesignEntry and implementing the virtual method *build* with OoRCScript. The DSL is *self-defining* and its syntax is based on message send as shown in figure 3.4. The principle is as follow:

- All objects in OoRCScript are instances of meta-descriptions. These object are denoted as description objects.
- The syntax is defined by sending a message (method) –denoted as description message– to a description object. This action will create a new description object based on the received message.
- Some built-in description objects are available for any design:
  - *input*,*output*,*inout*: for port declaration.
  - *signal,var* : for signal and variable declaration.
  - *arch*: for architecture definition.
  - *function, procedure*: for DSL function and procedure definition.
  - *alias*: create a reference to others signals/ports instead of creating new one.

OoRCScript can be executed normally inside the Pharo environment without the need of a dedicated compiler. The code can be verified (at run-time) as an interpreted language. The language is extensible by adding more description methods to meta-description classes. These methods can then be used directly in OoRCScript without any further modification.

OoRCScript code when executed, will create and connect all description objects to model a (sub)circuit. Although at this point the DSL code is syntactically correct, sometimes, its semantics are erroneous which makes the generated model improper. Therefore, it is worthwhile to perform an integral check of the circuit-model. For instance, the meta-model is able to detect some common problems: (1) assignment to signals of different data types, (2) sequential statements outside of process/procedure/function or using combinational statements inside a process, etc., (3) illegal operations on the data type of a signal, or (4) multi-source driving to a signal. In addition to this verification, in this phase, the system also the automatic signal resizing in the operations of two or more *LogicVector* signals of different size.

3.4 Coarse-grained Modeling: Hardware System Level Design Using Object Oriented Technique

Integration-based design approach involves two main steps: *system-level design* and *content based design*. The first step describes the system at conceptual level. It generalizes the system and specifies the relation between components without any detail of implementation. This allows to build an overall system specification. In doing so, a specific design problem can be generalized to have a more generic solution. This promotes the reuse, maintainability and integrability of the system in different application contexts with minimal modification and design effort. OO technique therefore plays an crucial role in system analysis and design. The second step consists of the implementation of the system for a specific application context. It is the traditional content-based design with respect to the overall system specification.

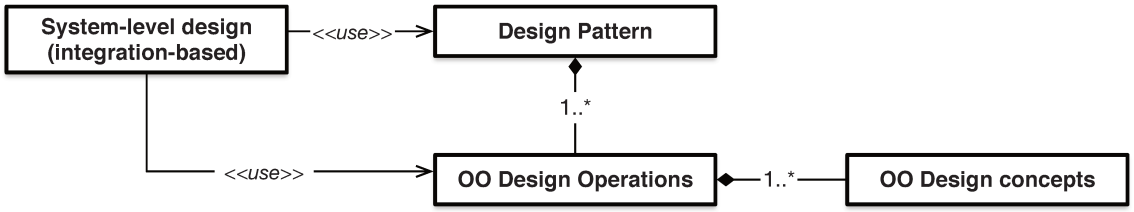


Figure 3.5: Using object oriented technique on HW system-level design

This section emphasizes the OO system-level design using our meta-model and DSL. Figure 3.5 shows the basic principle: System specification is described using a combination of *OO design operations* and *design patterns*. Each design operation, in turn, relies on a set of *basic OO concepts*.

3.4.1 Basic OO Concepts for HW Design

3.4.1.1 Structures Reuse

OoRCScript supports reuse in a design by describing commonly used structures in a separated method. Furthermore, for compatibility purpose, the language also supports VHDL functions and procedures. This can be done by following the syntax defined in listing 3.2.

```
1 <func/proc. name> [<param>:<value>]*
2   |<variable list>|
3   ^function/procedure is:{
4     <variable initialization >
5     <sequential statements>
6   }
```

Listing 3.2: Function/procedure definition

```
1 <Design class> map:{
2   [#<src port> -> <des port>.]*
3 }
```

Listing 3.3: Design reuse

A circuit model can be used as sub-circuit inside a design by sending the message *map:* to the corresponding design class as shown in listing 3.3. The parameters, in this case, are the port maps, which specify the connections between two designs. This will create an instance of *HDLComponent*.

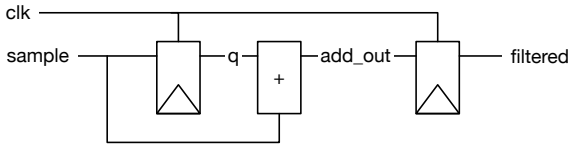
3.4.1.2 Inheritance

Like software classes, to increases the reusability, circuit designs using OoRCScript also supports inheritance. This latter allows the adaptation and extension of existing hardware designs to a wider context of application. By subclassing an existing design, child designs can inherit not only the interface but also all structures and behaviour of their parent. In addition, they are freely to add more ports to the interface or more logic to the architecture without changing the original design. However, it is the responsibility of designers to ensure the functional correctness of the new design where the architecture is extended. The meta-model only makes sure that there are no semantic incoherence in the design.

3.4.1.3 Polymorphism

Polymorphism allows some part of the design's architecture can be defined differently. This can be done by subclassing a design and redefine the desired structure. Redefinition (override) is supported in OoRCScript but is restricted. Only the entire architecture and reuse methods (OoRCScript methods and VHDL likes functions/procedures) are allowed to be redefined. This restriction ensures the semantic coherence of the circuit. Listing 3.4 shows an example of architecture override. The class *TwoTapFIR*, subclass of *SimpleFIR*, reimplements the entire architecture of the circuit by using an *DFF* (D Flip Flop - for introducing a delay between x_n and x_{n-1}). This introduces a more optimized solution for the circuit architecture.

In hardware design, polymorphism can be used to adapt an existing design to a new application context. A part of the design can be reused while the other part may be redefined to be compatible with the new context.




```
1  SimpleFIR subclass:#TwoTapFIR >> build
2    super build.
3    "override the entire architecture"
4    arch name:#arch override:{
5      signal  name:#(add_out q) is:(Unsigned size:32).
6      DFF map:{#d-> sample. #clk->clk. #q->q}.
7      add_out <- q + sample asUnsigned.
8      {clk} onChange:{
9        clk posedge ifTrue:{ filtered <- add_out asLogicVector }
10     }
11   }
```

Listing 3.4: An optimized re-implementation of *SimpleFIR* using inheritance and override features

3.4.1.4 Abstraction

Abstraction enables the definition of HW designs at a very conceptual level without any architectural detail. It provides an overall view of design's functionality while suppressing the details below the current level. Thus, an abstraction design can be used as a common interface for a family of circuits with different architectural and/or behavioural implementations but which share the same external view. For example, in a Master/Slave bus design, the bus controller may be interested mainly in the way masters/slaves are controlled rather than how they are actually performed. In this case, all masters/slaves can be generalized into a single abstract master/slave design.

OoRCScript allows to define *virtual* (abstract) architecture or methods (OoRCScript method, VHDL-like functions/procedures) in a design. This kind of design does not represent an actual circuit, but instead it materializes a template. Sub-designs have responsibility to implement the virtual architecture or methods.

In listing 3.5, the class *SimpleFIR* defines an abstract architecture. An actual architecture must be implemented in all subclasses (*e.g. TwoTapFIR*) of this class. Listing 3.6 shows another possibility of abstraction using an OoRCScript method. The architecture is set as a return of the *buildArch* method. This is an abstract method. All subclass must implement it by specifying the architecture's descriptions.

```
SimpleFIR >> build
  "inputs/outputs definition"
  ...
  arch abstract:#arch
```

Listing 3.5: Abstract architecture

```
SimpleFIR >> build
  "inputs/outputs definition"
  ...
  architecture name:#arch is:self
      buildArch
"virtual method definition"
SimpleFIR >> buildArch
  ^self subclassResponsibility
```

Listing 3.6: OoRCScript abstract method

Abstraction is ideal to perform system level design. It provides an overall view of the conceptual system by specifying how circuit modules are connected and interact with other. Detail implementation can be realized later depending on the use context of the system.

3.4.2 Basic OO Design Operations

Design operations are driven by basic OO concepts. They strengthen the system specification by maintaining the clarity of the design hierarchies. Basically, there are 4 main design operations:

1. GENERALIZATION: As defined by the name, generalization is the process of extracting share characteristics of two or more designs and combining them into a generalized design. Share characteristics may be external interface or internal structures. This operation aims at generalizing a design problem or defining a generic family of similar designs. Generalization uses the inheritance and abstraction concepts.

2. SPECIALIZATION: In contrast to generalization, specification reduces the application context of existing designs by adding more specific features to the original design. It promotes the extensibility and reusability of existing designs. Specialization uses inheritance and polymorphism concepts.

3. REALIZATION: Provides a content-based implementation of a design. Unlike specialization, –which relies on existing designs– realization specifies the implementation of a design from scratch. This design could be a brand new design using OoRCScript or an implementation of existing abstracting design.

4. COMPOSITION: this operation allows to embed one or more low-level design to a higher-level design. It aims at performing hierarchical system design. It uses the previously presented design reuse concept.

3.4.3 OOD Pattern on Hardware Design

Design operations, once combined together, can specify the relationship of system components and thus allow to construct the system specification. Some frequently used combinations –denoted as *design*

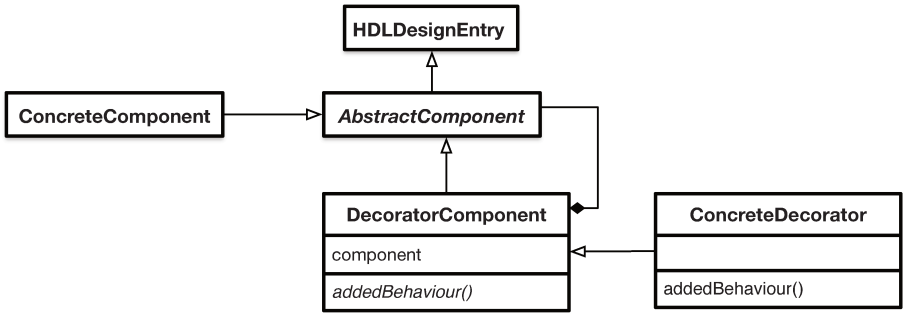


Figure 3.6: Base structure of decorator pattern applied on hardware design

pattern– can be predefined and documented to facilitate the system-level design process. This section presents some software patterns [GHJV95] that can be used on hardware design using our meta-model. However, the design process is not limited to these patterns and can rely on any possible combination of design operations.

3.4.3.1 Decorator

Intent attach additional responsibility to an existing design (e.g. communication interface design). Decorator allows reuse of existing component and adapts it to new application context. It is alternative to subclassing but indeed is more flexible, since it allows the additional functionality can be dynamically added to the circuit model at run-time, depending on the target component.

Structure figure 3.6.

Operations used abstraction, composition, specialization and realization.

Description Both decorator component (e.g. *DecoratorComponent*) and the target component (e.g. *ConcreteComponent*), share the same external interface by subclassing an architectural abstract component (e.g. *AbstractComponent*). The decorator embeds a concrete implementation of the abstract design and extends the external interface if needed. It enhances the functionality of the target component by provide a abstract method (*#addBehaviour*). All realization of the decorator (e.g. *ConcreteDecorator*) has the responsibility to implement this method for added features. Note that, the *ConcreteComponent* provides only a specific implementation of the abstract design without extending the interface.

Example listing 3.7 shows an example of adding a simple pixel counter to an existing pixel filter (HSV filter or RGB filter), demonstrated by figure 3.7. The *PixelFilter* filters a pixel (24 bits RGB value) using a color pattern (threshold value). The added counter will count the number of pixels that pass the threshold value.

```
1  "Abstract component defining a pixel filter based on a color pattern"
2  HDLDesignEntry subclass:#PixelFilter >> build
3    input name:#(clk reset start) is:Logic.
4    input name:#(pixel color_pt) is:(LogicVector size:24).
```

```
5    output name:#(flag done) is:Logic.
6    arch abstract:#arch.
7  "Two concret components implementing two filters: RGB filter and HSV filter
       "
8  PixelFilter subclass:#RGBFilter >> build
9    super build.
10   arch name:#arch is:{..."filter process here"}
11 PixelFilter subclass:#HSVFilter >> build
12   super build.
13   arch name:#arch is:{..."filter process here"}
14 "A decorator component add an extra functionality to existing filter"
15 PixelFilter subclass:#ExtraFilter >> newFrom:aFilter
16   filter := aFilter
17 ExtraFilter >> build
18   super build.
19   arch name:#arch is:{self extra}
20 ExtraFilter >> extra
21   self subclassResponsibility
22 "A concret decorator that adds a pixel counter to existing Filter (RGB
       filter or HSV filter)"
23 ExtraFilter subclass:#FilterWithCounter >> extra
24   output:#pxcnt is:Integer.
25   signal name:#(s_flag s_done) is:Logic.
26   filter map:{#clk->clk. #reset->reset. #start->start. #pixel->pixel.
         #color_pt -> color_pt. #flag->s_flag. #done->s_done}.
27   done <- s_done.
28   flag <- s_flag.
29   {reset. clk} onChange:{
30     var name:#cnt is:Integer.
31     reset ifTrue:{cnt <- 0} ifFalse:{
32       clk posedge ifTrue:{
33         (s_done = true and:(s_flag = true)) ifTrue:{
34           cnt <- cnt+1.
35     }}}.
36     pxcnt <- cnt
37   }
```

Listing 3.7: Example of adding a counter to an existing pixel filter

3.4.3.2 Adapter

Intent Convert the interface of a design into another interface expected by the system. This pattern allows to plug an existing design of different interface into the system by using an commonly known interface.

Structure figure 3.8.

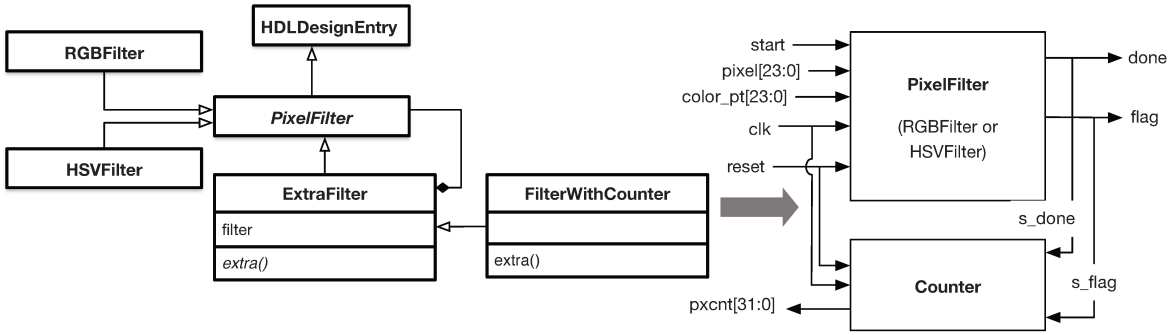


Figure 3.7: Example: attach more responsibility to a pixel filter using decorator pattern

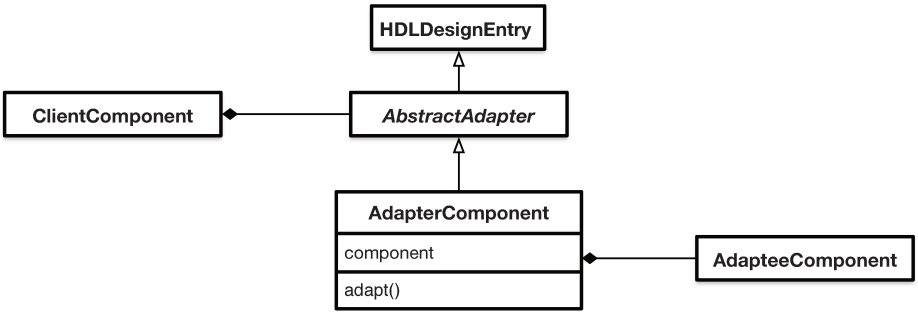


Figure 3.8: Base structure of adapter pattern applied on hardware design

Operations used abstraction, generalization, composition.

Description The commonly used interface is defined by an abstract design (e.g. *AbstractAdapter*). The adapter design (e.g. *AdapterComponent*) will realize this interface by composing the target component (e.g. *AdapteeComponent*) and perform interface conversion (*#adapt*). Unlike the software version, in this pattern the relation *client/adapter* and *adapter/adaptee* is stronger (composition).

Example Listing 3.8 shows a simple implementation of interface conversion demonstrated in figure 3.9 using adapter pattern. In this case, a client collects pixel data from a serial connection (UART). The client wants to pass data to the pixel filter (i.e. *PixelFilter* from previous example) by sending separately each R,G,B (8 bits) components of the pixel and the color pattern. The filter, however, accepts only 24 bits color value. In this case, we use an adapter (defined by *AbstractAdapter*) to provide conversion between the two interfaces.

```
1  "Abstract interface known by client"
2  HDLDesignEntry subclass:#AbsractAdapter >> build
3    input name:#(PR PG PB TR TG TB) is:(LogicVector size:8).
4    input name: #(start clk reset) is:Logic.
5    output name:#(flag done) is:Logic.
6    arch abstract:#arch.
7  "Adapter component implement an interface conversion"
8  AbstractAdapter subclass:#RGBValue2Pixel >> buildFrom:aFilter
9    super build.
10   arch name:#arch is:{
```

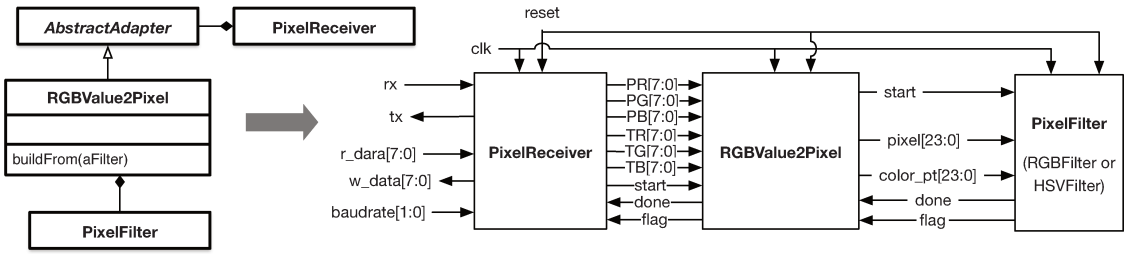


Figure 3.9: Example: Interface conversion using adapter pattern

```
11      signal name:#(px pattern) is:(LogicVector size:24).
12      px <- (PR,PB,PG).
13      pattern <- (TR,TG,TB).
14      aFilter map:{#clk->clk. #reset->reset. #start->start. #pixel->px.
            #color_pt -> pattern. #flag->flag. #done->done}
15    }
16  "Client component that use an abstract adapter"
17  HDLDesignEntry subclass:#PixelReceiver >> buildFrom:adapter
18    input name:#(clk reset rx) is:Logic.
19    input name:#r_data is:(LogicVector size:8).
20    output name:#w_data is:(LogicVector size:8).
21    input name:#baudrate is:(LogicVector size:2).
22    output name:#tx is:Logic.
23    arch name:#arch is:{
24      signal name:#(PR PG PB TR TG TB) is:(LogicVector size:8).
25      signal name:#(start done flag) is:Logic.
26      adapter map:{#clk->clk. #reset->reset. #start->start. #done->done.
            #flag->flag. #PR=>PR. #PG=>PG. #PB=>PB. #TR=>TR. #TG->TG. #TB=>TB}.
27      "UART data encode/decode here"
28      ...
29    }
```

Listing 3.8: Implementation of interface converting using adapter pattern

3.4.3.3 Composite

Intent Compose components into a hierarchical structure. It is ideal for designing hierarchical system or system with generic (recursive) data path (e.g. data/address IO bus interface, registers, etc.). Note that the hierarchical composition of the system can be performed dynamically at run-time of the circuit model.

Structure figure 3.10.

Operations used abstraction, composition, specialization and realization.

Description This pattern is composed of two main components –which share the same base interface (e.g. *Component*): an atomic component (e.g. *AtomicComponent*) and a composite component

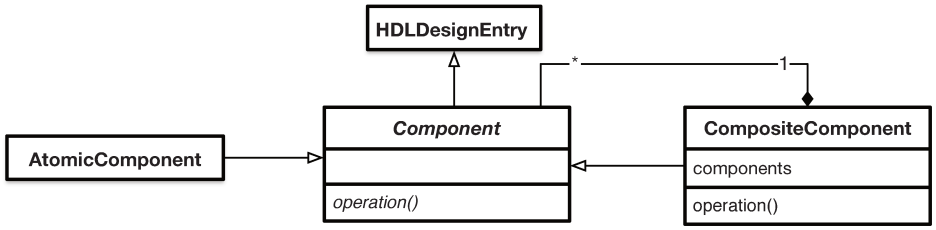


Figure 3.10: Base structure of composite pattern applied on hardware design

(e.g. *CompositeComponent*). The first component provides an implementation of the generic interface while the second compose a group of components shared the same interface. The composite component may extend the base interface and add (*#opearation*) more behavioural functionalities (e.g. arbiter for a master/slaves bus etc.).

Example Section 3.5.4 will detail a use-case of composite pattern in HW design.

3.4.3.4 Bridge

Intent Decouple the interface from the architectural implementation so that they can vary independently depending on different application contexts. Although the meta-model supports multiple-architectures definition of a design, this pattern is more flexible. It promotes the independent change/evolution of the interface and the implementation.

Structure figure 3.11.

Operations used abstraction, composition, specialization and realization.

Description The interface is defined by an architectural abstract design (e.g. *AbstractionComponent*). This design delegates its implementation to a separated implementor (e.g. *Implementor*). There may be many implementors (e.g. *ConcreteImplementor*) depending on different application contexts. The abstract design only specifies how the implementor is connected to the interface via the *#connectImplementor* method. All subclasses of this design have the responsibility to implement that method. In doing so, the interface of the design can be freely extended (e.g. *RefinedComponent*) while always remaining compatible with any implementor. Also, additional logics can be easily added to the refined design's architecture for functional extension.

Example Listing 3.9 shows an example implementation of bridge pattern demonstrated by figure 3.12. The implementors (*RGBFilter* and *HSVFilter*) are separated from the difenrent interfaces (UART - *FilterFromUART* or I2C - *FilterFromI2C*) using to collect pixel data. They are free to vary while remain compatible with any interface/implementor.

```
1 "Abstract interface refers to an implementor"
2 HDLDesignEntry subclass:#AbsractComponent >> buildFrom:aFilter
3   filter := aFilter.
4   input name: #(clk reset) is:Logic.
5   arch name:#arch is:{
```

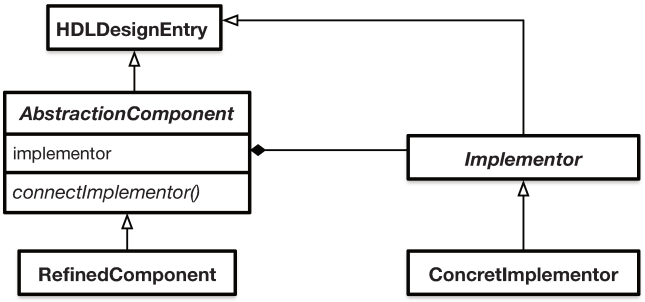


Figure 3.11: Base structure of bridge pattern applied on hardware design

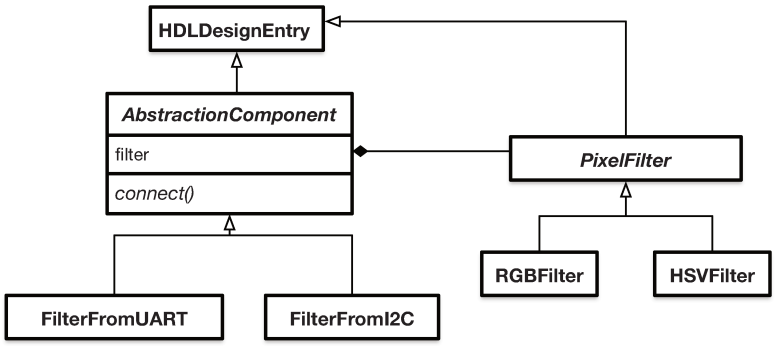


Figure 3.12: Example: Interface & architectural implementation decoupling using bridge

```
6       signal name:#(px pattern) is:(LogicVector size:24).
7       signal name:#(start done flag) is:Logic.
8       filter map:{#clk->clk. #reset->reset. #start->start. #pixel->px.
            #color_pt -> pattern. #flag->flag. #done->done}.
9       self connect.
10    }
11  AbstractComponent >> connect
12    self subclassResponsibility
13  "Refined component 1: Get pixel data for filtering using UART interface"
14  AbstractComponent subclass:#FilterFromUART >> connect
15    input name:#rx is:Logic.
16    input name:#r_data is:(LogicVector size:8).
17    output name:#w_data is:(LogicVector size:8).
18    input name:#baudrate is:(LogicVector size:2).
19    output name:#tx is:Logic.
20    ... "UART data encode/decode here"
21  "Refined component 2: Get pixel data for filtering using I2C interface"
22  AbstractComponent subclass:#FilterFromI2C >> connect
23    inout name:#(sioc siod) is:Logic.
24    ... "I2C data encode/decode here"
```

Listing 3.9: Implementation: decouple data interface (UART, I2C)

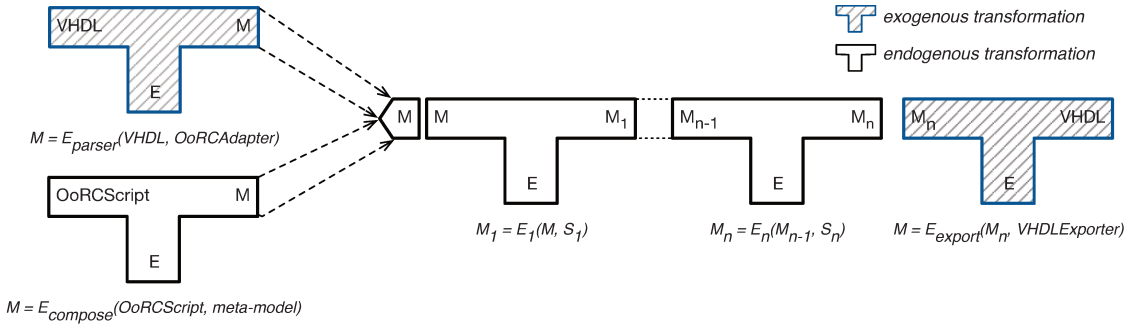


Figure 3.13: T-diagrams represent basic model transformations supported in OoRC

3.5 Circuit Model Transformation

3.5.1 Overview of the Transformation Process

Given a circuit model M, a transformation E of M to an output model M' can be defined as a function of:

$$M' = E(M, S) \tag{3.1}$$

In where S is a set of constraints, specifications or a template, etc. The function E defines an automatic transformation operation (refactoring, integration, exporting, etc.). It processes the model M according to the input S to produce M'.

OoRC provides a complete object oriented solution for model transformation. The meta-model considers all circuits structures as objects. A circuit model is an object composed of child objects connected together. These objects represent the internal architecture of the circuit. The function E can be defined to work directly on them and can benefit from object oriented techniques for model processing. By using live objects, we can take advantage of the dynamic environment for the evolution of the circuit model at run time. The verification of circuit model can be realized progressively and automatically during the design process, since the model holds not only the structure but also the behaviour of the circuit.

All basic model transformation supported by OoRC can be visualized by using a T-diagram as shown in figure 3.13. Basically Tombstone diagrams are used to illustrate the functioning of translator systems. A common use case of these diagrams is to demonstrate the compiling process of a source language (left of T) to a target language (right of T) using a compiler (middle of T). Here we apply this concept to present a *"transformation E (middle of T) of an input model M (left of T) to an output model M' (right of T)"*.

In OoRC, *VHDL-to-model* (import) and *model-to-VHDL* (export) transformations are classified as *exogenous transformations*. All others transformations are *endogenous*, since they are expressed using the same meta-model.

Traditionally, an input model M will be processed by the transformation E to produce a separated output model. In our system, in additionally, the circuit model itself is reflective. That is, the transformation E can be defined inside the model. The model is able to modify its own structure/behaviour at runtime and flexibly adapt to different application context. This is useful when expressing the progressive evolution of the model. A use case of this feature is when the meta-model is used to build graphic

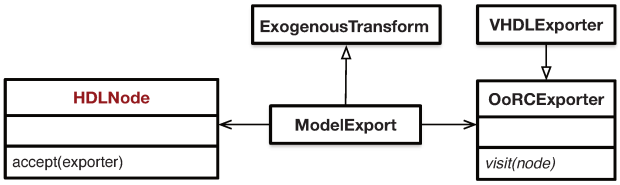


Figure 3.14: By using a *visitor pattern*, a circuit model can be independently exported to other model (VHDL/Verilog, etc.)

CAD tool. An empty circuit model is initialized at the beginning. As long as user graphically adds or connects components, it will restructure itself to update new changes.

```
1 |model|
2 model := SimpleFIR new build.
3 (model ports, model architecture signals)
4     collect:[:e| e data size = 32]
5       thenDo:[:e| e data resize: 16].
6 model performIntegrityCheck.
```

Listing 3.10: Create a new FIR filter model and resize all data signals to 16 bits

The transformation E can be implemented using APIs functions of the meta-model. Listing 3.10 shows a simple transformation of the previous FIR filter model by collecting all 32 bits signals and resize them to 16 bits. The transformation is committed to the model itself. Usually, an integrity check of the output model is needed after the transformation to ensure its correctness.

3.5.2 Exporting Circuit Models

OoRC provides a flexible exportation mechanism for converting circuit models to external models (i.e. models described by other meta-models) as shown in figure 3.14. The system introduces an abstract exportation interface to different target meta-models using a visitor software pattern. By subclassing *OoRCAdapter* and implementing all *visit* methods, any meta-description object can be independently transformed to an equivalent model of a target meta-model. This allows an independent coupling between the exporter and meta-descriptions. New exporters can be defined and plugged into the system without changing the OoRC meta-model. The *VHDLExporter* is a vivid example, it converts an input circuit model to an equivalent VHDL representation. This class is used in our simulator to perform an external simulation or in our synthesis toolset for low level synthesis on actual hardware (section 3.7).

3.5.3 Legacy VHDL Reuse via a Dedicated VHDL Parser

One common problem when modeling hardware systems is how we can reuse existing VHDL designs in our model. Such reuse can enrich the models design while reducing the production cost by benefiting of an existing rich set of third-party VHDL code. Thanks to the dedicated VHDL parser, our meta-model supports the reuse of almost every VHDL designs conforming to the IEEE 1076.66 RTL standard. The parser can be considered as an implementation of the model transformation function E in equation 3.1. It performs a *text-to-model* transformation.

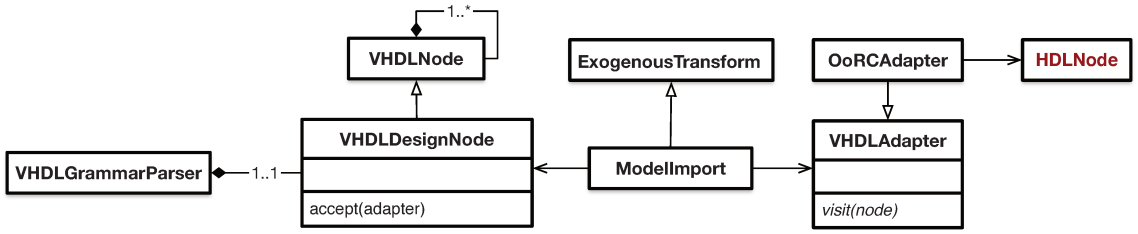


Figure 3.15: VHDL designs are parsed into parser trees before being converted to circuit models using *OoRCAdapter*

3.5.3.1 A dedicated VHDL parser

Our parser is built based on the ANTLR VHDL Grammar version 4 [PF11]. Since all rules have been implemented, the parser is able to parse any VHDL structures, either synthesizable or simulation structures. However, to build circuit models from VHDL designs, only synthesizable structures are enable.

As shown in figure 3.15, to import a legacy design into the system, the VHDL files are first parsed by *VHDLGrammarParser*. This class implements all VHDL grammar rules and helps build a parser tree from input VHDL files (when succeeded). The parser tree is an independent tree of *VHDLNode* objects. It can be converted to any model by using a *visitor pattern*. By subclassing *VHDLAdapter*, one can define as many as possible adapters (visitors) for model transformation. This allows to separate completely the parser with the rest of the system (meta-model). Therefore it can be used individually in other projects without any problem. In OoRC, the *OoRCAdapter* appears as a "plugin" to the parser that helps initialize a circuit model from VHDL design.

VHDL designs, once imported, will be represented as a subclass of *HDLDesignEntry*. They can be modified by editing the DSL code in the *build* method. A subclassing to one of these classes allows to easily extend the design.

3.5.3.2 Validity of the Parser

The VHDL parser plays an important role in the framework since it is used to import VHDL legacy code into our system. It is worth being checked carefully. For this purpose, it has been tested against some sets of VHDL files used as benchmarks:

- ANTLRset[PF11](13files) which contains the standards VHDL packages like: *std_logic_1164*, *numeric_std*, *arith*, *textio*, etc.
- The IWLS 2005 Benchmarks [Alb05] which contains diverse circuit designs coming from open source community of hardware designers and industry to represent a variety of applications. Since this set is in VHDL, Verilog and BLIF, we use only the VHDL files available in two subsets: (1) the ITC'99 benchmarks [CRS00] (30 files) and the Gaisler Research set (242 files).

Since the parser implements all VHDL grammar, it passed these benchmarks without any problem. Note that this test only verifies the parser which reads VHDL codes and constructs a parser tree of them. The conversion of the abstract syntax tree into a circuit model using the meta-model is much more complex. Such validation involves the research work on the model-checking domain. So, we

left this test as future work. Moreover, the test requires proper and well-defined synthesizable VHDL benchmarks [‡].

3.5.4 Automatic Circuits Integration and Configuration

Automatic circuits integration and configuration provide a mechanism to put designs together and enable a compatible communication between them via a common interface. OoRC supports this by introducing *HDLDynamicCompositeDesign*[§], a dedicated class for designs integration as illustrated in figure 3.16. This class allows to compose a set of input designs and wrap them in a common interface. This interfacing-process is semi-automatic and can be defined as a function of: $M_I = E_{composite}(D, S)$ where $D = \{D_i | i = 1..n\}$ is a set of input designs and $S = \{S_i | i = 1..n\}$ is a set of corresponding *ports classifications* (for each input design). A ports classification S_i of a design D_i specifies how a port of D_i will be connected to the interface. Basically, all ports can be classified into one of following categories:

- *clock,reset*: clock and reset ports, these ports will be connected directly to the *clock,reset* port of the interface.
- Control ports: *start,end, etc.* control the behaviour of the circuit. These ports will be connected to corresponding internal signals of the interface for manual processing. A design may or may not have these type of ports.
- *Physical ports* are ports which need to be forwarded directly to the interface inputs/outputs without any processing. They may be used to connect to an actual hardware device (sensors, physical bus, etc.). These ports will be connected directly to corresponding input/output ports (also denoted as *forwarded ports*) of the interface. They are optional in a design.
- *Logical ports*: address/data and other control ports that need manually wired inside the interface to adapt the design to the new interface.

Since it's a semi-automatic process, the $E_{composite}$ function will be performed in two phases:

Automatic pre-processing : in this phase, an empty composite model is first initialized. The model then generates all necessary signals/ports and connects them to corresponding input design D_i based on their port classification S_i.

Manual processing: Additional ports of the interface are manually defined in this phase. The model then execute the method *craft* to perform the internal linking of *logical signals* (connected to logical ports in previous phase) and control signal (*start,end*) of each input design D_i to the interface (red zone in figure 3.16). In *HDLDynamicCompositeDesign*, this method is a virtual method. The class is not used directly, it must to be subclassed. All its subclasses have the responsibility to implement the method *craft* to specify internal linking logic depending on each interface.

Predefined Interface for IP Integration

The dynamic composite design method can be used to build a predefined interface for IP-integration. This can be considered as an interface template. Designers can use it to implement their application

[‡]The Parser can work with either simulation or synthesizable VHDL code, but the meta-model supports only synthesizable VHDL structures

[§]This class is based on the hardware *composite pattern*

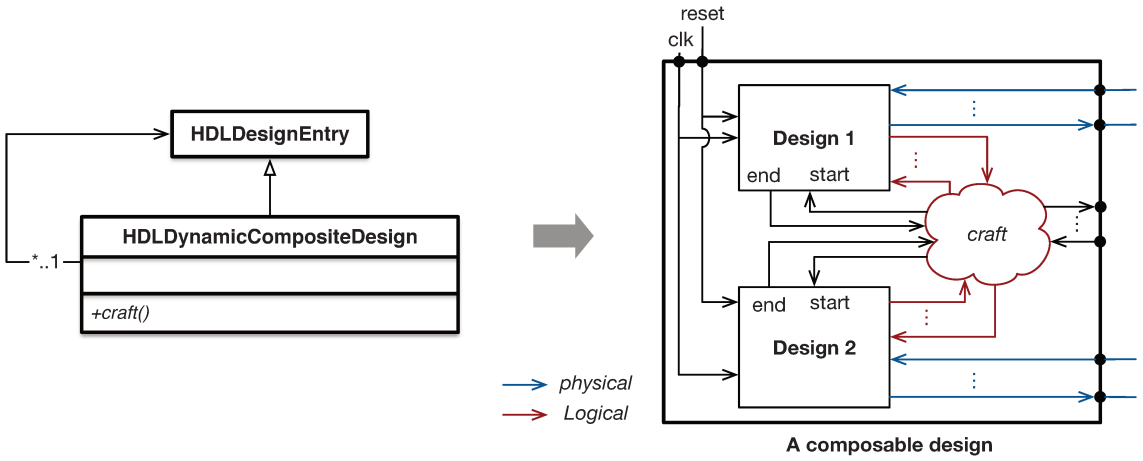


Figure 3.16: *HDLDynamicCompositeDesign* allows *semi-automatic* composition of models and provides a virtual *process* method for manual models linking

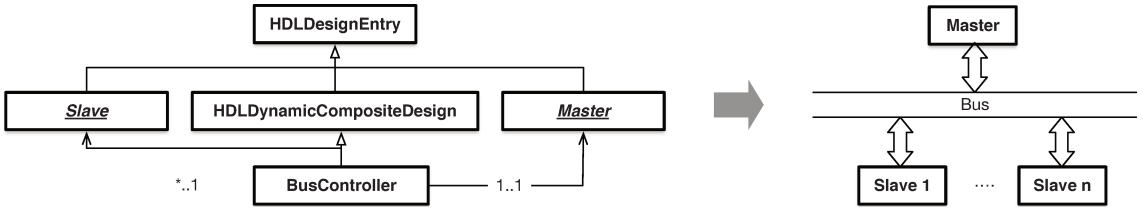


Figure 3.17: A predefined Master-slave bus interface: IP designers just need to subclass Slave or Master to define their application logic

modules. These modules can then be plugged automatically to the interface without any further modification. Figure 3.17 shows an example of a Master-Slave bus interface definition. The *Slave* and *Master* classes are architectural abstract designs. They define only common *Master/Slave* external interface (ports) without internal architecture. The *BusController* is a composite design of a *Master* and one or more *Slaves*. It defines how a *Slave* can be active based on the *Master* (addressing/arbiter. etc). This technique provides only a generic *Master/Slave* bus interface to IP designers. By subclassing *Slave* and implementing the architecture, they can define any kind of slaves needed. The same method can be used on *Master*, there are maybe different masters for connecting to different physical interfaces (serial/parallel, etc.). Obviously, subclasses of *Slave/master* can define their own additional ports. These ports will be classified as *physical* and will be forwarded (by the *BusController*) to the bus interface for external connection. With this template, the slave/master architecture can freely vary while always being compatible with the bus and the communication protocol. Chapter 4 will detail how to use this approach to build a generic Wishbone interface template or to automatically generate a hardware debug circuit for any input design.

3.5.5 Discussion

A *Model-to-VHDL* transformation is mandatory to translate the circuit model to VHDL for bitstream synthesis while *VHDL-to-model* is necessary to add backward support to OoRC. This feature is important seeing that the majority of existing HW libraries are based on traditional HDL model. Not many

system-level design methodologies support this feature in their system. Some approaches (e.g. IP-XACT [IEE14]) allow this but require an additional manual IP decoration/meta-descriptions step, then work on these descriptions/decorations rather than on the IP itself. Therefore, they are hard to deeply commit change to the IP (optimization, restructuring, etc.). On the other hand, OoRC, allows to fully import legacy HDL IP as a native model, that one can perform any transformation on it lately.

Automatic circuits integration and configuration provides a (semi)-automatic mechanism for combining several arbitrary designs together via a common interface template. This feature uses design pattern and performs some correct-by-constructions and automatic structure generation of the interface depending on the input set of designs. Advantage of this approach is that it abstracts away and separates the interface from the real implementation so that it can be independently applied on different application contexts. This reduces the design cost for the interface which alway remains as a time-consuming and less-contributive task.

3.6 "In-vivo" Circuit Models Simulation

3.6.1 Execution Model: time-driven vs. event-driven

To understand how the simulated execution of FPGA circuits works, one must understand what kind of execution model is used in the meta-model. For such a model, two kinds of systems need to be taken into account, the *continuous systems* and the *discrete systems* [Slo15]. In the first ones, the state of the system (signals, ports) changes continuously with respect to time, whereas in the latter ones, the state changes instantaneously at separate points in times. In reality, there are few systems that are either completely continuous or discrete, although often, one type dominates the other. For example, a synchronous circuit that uses the global clock can be considered as a continuous system since its state can be changed at each clock. But when we consider the system at gate-level, when a part of the circuit is active, all related operations will be performed and make change on its outputs. This change, in consequence, will trigger instantaneously other parts connected to it. This process is repeated until the state of each part becomes stable. These parts, therefore, can be considered as discrete systems. The challenge here is to find a computational model that mimics closely the behaviour of such time-advance systems. There are, in fact, two models that can be used in this case: *time-driven* and *event-driven*.

A continuous system can be easily simulated using the *time-driven* [PL08, Slo15]. With this approach, the simulation advances time with a fix increment of exactly Δt time units which is called simulation clock[¶]. After each clock, the state of the system is updated for the interval of $[t, t + \Delta t]$. This approach, however, is not very appropriate for simulating a discrete system. For a such system, the time step Δt must be small enough to capture all events. Often, this time step is extremely small which is unacceptable as the simulation time involved. Furthermore, there are obviously empty time steps that cause wasting simulation time.

An *event-driven* simulation [PL08, Slo15, SC95] has a nature close to a discrete system. The simulation time in this case advances directly to the next-event time. An event represents a state change of the system caused by incoming data or internal processes. For the case of a discrete system, the approach

[¶]The simulation clock is unrelated to the hardware clock and is used only by the simulator to keep track of the simulation time as the simulation proceeds

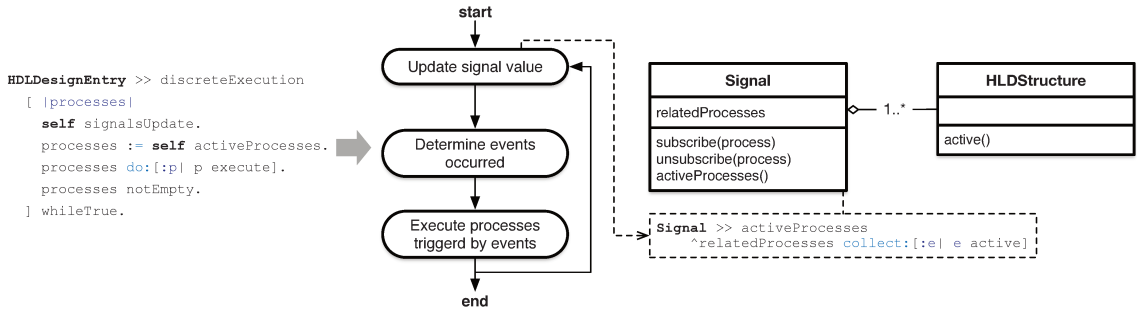


Figure 3.18: Event-driven simulation of discrete system using an observer pattern. Assignment of a value to a signal will cause all related processes become active.

consists of following steps: (1) the simulation time is initialised and all the occurrence times of future events are collected in an *event-queue*. (2) The simulation time is advanced to the closest *next-event* time in the *event-queue*. (3) The state of the system is updated by executing the scheduled events. (4) The *event-queue* is updated and the second step is repeated. The advantage of this method is that we can skip the periods of inactivity by advancing directly to the next event time and therefore, avoiding unnecessary empty cycles. In term of causality, this is perfectly safe because the state change of the system only happens at event times. This model of simulation can be easily applied to a continuous system since it is based on the occurrence of events and in such continuous system, each event occurs after a Δt interval of time.

This description shows why we use the *event-driven* simulation model in OoRC.

3.6.2 Event-driven Simulation of Circuit Models

As mentioned, a circuit may contain many combinational and process parts. A process has a sensitivity list consisting of signals that will trigger it as their values are changed. In our meta-model, all combinational parts are considered as *in-line* processes in which the inputs of each part are its sensitivity list. To trigger the processes we use the observer pattern. Each time a signal changes its value, it will announce an event to all processes that have it in their sensitivity list. Figure 3.18 presents the basic idea of this approach.

Recall that each signal is an object with history, and it only takes the new value after being updated. In the signals update phase, any change in a signal (`#signalsUpdate`) will activate all processes watching it. Since these processes are performed in parallel and the state of the circuit doesn't change until the next signals update, their execution order doesn't matter. An execution of a process at a point simulates its behaviour based on current input (signals) values. This action may make change on some signals which in turn, will trigger other processes. This propagation execution is repeated until all signals stabilise and there are no more active processes.

Based on this discrete execution, we can model the complete circuit execution in responding to the incoming data. OoRC supports two modes of execution depending on whether the timing information of is important or not:

Functional simulation: in this mode, the circuit model acts as a blackbox. All input ports of the circuit model are first assigned to new values. The execution will then be performed until a specific condition

is met. At that point, the outputs of the model can be inspected. This kind of simulation, as explained by its name, interests only how outputs values of a circuit change depending on its inputs, regardless of the time needed to complete the calculation. This is helpful for software/hardware co-development. Developers can use the circuit model as an hardware abstraction to build and test their software without worrying about the actual hardware implementation. Listing 3.11 shows an implementation of this simulation mode.

```
HDLDesignEntry >> execUntil:aCond
  aCond whileFalse:[
    self clock ifNotNil:[
      self clock toggle].
    self discreteExecution.
  ]
```

Listing 3.11: Functional simulation implementation

```
HDLDesignEntry >>
    execWith:queue dumpOn:stream
  |candidate|
  [
    candidate := queue nextEvent.
    candidate notNil
  ] whileTrue:[
    self assignSignals:candidate
       signals.
    self discreteExecution.
    self snapshotAt:(candidate time)
             on:stream
  ]
```

Listing 3.12: Behavioural simulation implementation

Behavioural simulation: this is the traditional hardware simulation. In this mode, the system keeps tracking the behaviour of the circuit during time. This behaviour can be defined by the intrinsic signal values of the circuit over time. To do this, the execution of the circuit model will be performed with a time queue. This latter contains the next-event times and signals that will be assigned to new values in each event, as illustrated in listing 3.12. We can consider this time-queue as a test-bench in traditional simulators. At each execution step, the simulation time advances to the closest next-event time, the inputs signals related to this event are assigned to new values and the propagation execution is performed. At the end of each event, the state of the circuit along with the current timing information is recorded in a Value Change Dump file (VCD). This process is repeated until the time-queue is empty. The final VCD file can then be viewed by a external VCD viewer for analyzing.

3.7 Interfacing the OoRC meta-model with External Tools

3.7.1 "Ex-vivo" Simulation Using an External Simulator

It is possible to simulate the circuit model with an external simulate . This can be done in an indirect manner by: (1) generating a test-bench from the time queue and (2) exporting the target circuit as Unit Under Test for the test-bench. The simulation can then be performed on the test-bench by invoking a series of external simulator's commands. This method applies only for *behavioural simulation*.

Currently, the system supports automatic "ex-vivo" simulation using the open-source simulator GHDL

3.7.2 Circuit model synthesis and deployment

To support the deployment of circuit models on an actual hardware, OoRC has a dedicated toolset for automatic synthesis. It can be performed in three main steps:

Firstly, it encapsulates the target model in a dedicated interface. This interface allows an agreed communication between the FPGA circuit and the physical world (e.g. processor, sensors, etc.). The final model (including interface) will be finally exported to VHDL for low level synthesis.

Secondly, it generates a device configuration for a selected hardware. This configuration is hardware specific and different between vendors. It specifies all the options needed for a low level synthesis on a particular device. The configuration also sets up the physical mapping of the interface generated on the previous step. This allows to connect circuit ports to actual FPGA IO ports.

Finally, the toolset performs a low level synthesis based on the VHDL code and the device configuration obtained from previous steps. This is done automatically by invoking a series of commands provided by vendor synthesis tools. If all commands are successfully executed, a bitstream will be generated at the end of the process and ready for deployment.

In the system, these steps can be modelled by three main concepts: *PhysicalInterface*, *DeviceConfiguration* and *Synthesis* as illustrated on top of figure 3.19. *PhysicalInterface* is a subclass of *HDLDynamicCompositeDesign*. It is a composite design which allows to define a generic physical interface wrapper (e.g. serial, parallel, usb, ethernet, etc.) for any input circuit model. This wrapper enables the communication of the circuit with the outside world (e.g. processor, network, etc.). The *DeviceConfiguration* class, in additional to device specification, also specifies how an interface is mapped to the FPGA IOs. The mapping configuration of an interface is separated from its definition to ensure the interface independent regard to different devices. When an interface (*PhysicalInterface*) *accept* a configuration (*DeviceConfiguration*), it requires the configuration to set up the ports map for it (via the method *portMap*). A new added device needs to define all *portMap* methods for the interfaces that it supports. Note that, *physical* ports (of the input design) forwarded by the interface are design specific and are undefined at the design time of the generic interface. In this case, they need to be manually mapped at run time by the configuration (using the method *manualMap*).

The interface and configuration are used by the *Synthesis* to generate all manifest files needed for a low-level synthesis. The system supports synthesis via SSH (using the class *RemoteSynthesis*). This allows all vendors tools (Xilinx, Altera, etc.) can be centralized on a server, and the synthesis will be performed remotely based on a selected vendor device.

The bottom of figure 3.19 shows a simple implementation of synthesis GUI tool from the class diagram. The list on the left displays all design classes available in the system. Users need to manually classify all ports for a selected design (*LGCameraUnit*). This decides how each port are connected in the interface. On the right, one can select an desired interface (*APF51Builder*) and a supported device (*APF51Imx*). Users may also need to specify the port map (*IO*) for all physical ports of the target design, since they are unknown by the interface at design time. Thanks to the dynamic environment of Smaltalk, when new design, interface, or device classes are added to the system, they will be updated automatically to the GUI.

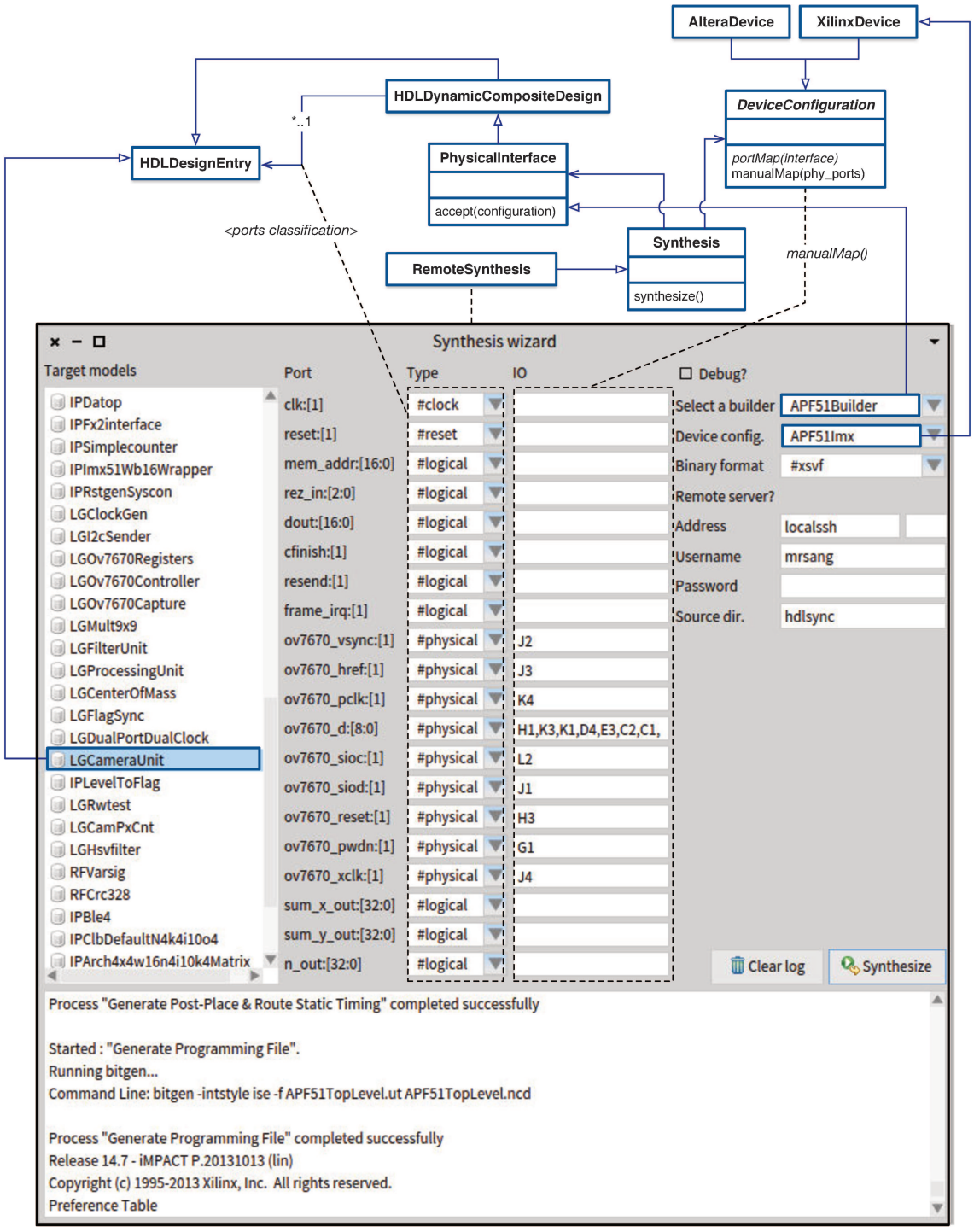


Figure 3.19: Class digram design and the corresponding GUI implementation of our model synthesis toolset

3.8 Summary

In this chapter, we have presented the OoRC meta-model, our first contribution that provides an object oriented approach for digital circuits modeling. The meta-model allows to describe a circuit as a graph of connected live objects. This graph is especially handy for model transformation/processing (refactoring, integration, etc.). A circuit model is executable an can perform a *in-vivo* hardware simulation relying on events. This work was partly published in [LLF+15].

One can manually design circuit models using the dedicated DSL. The DSL brings oriented-object technique to hardware description to support system-level design and enables efficient reuse of IPs. It is also possible to reuse third-party hardware designs (in VHDL) in our system thanks to the built-in VHDL parser.

The chapter focused mainly on basic concepts and methodologies of OoRC. The meta-model covers all basic aspects of digital hardware design, from hardware description and simulation to automatic low level synthesis. Based on this solution, high level (CAD) tools can be built for circuit design or for automatic circuit model processing. Chapter 4 will show another possible use of the OoRC meta-model for abstracting hardware circuits and automatically handling software/hardware communication.

You can mass-produce hardware; you cannot mass-produce software - you cannot mass-produce the human mind.

Michio Kaku

4

OoRCBridge: Seamless Integration of FPGAs with High-Level Software

Contents

This chapter presents OoRCBridge, our dedicated middleware and toolset for software/hardware interfacing. It aims at being a generic solution for integrating FPGA within existing software systems. Middlewares interoperability is supported in our system so that programmers can stay on their middleware while having a possibility of hardware (FPGA) interaction. Section 4.1 provides an overview of our methodology. Section 4.2 describes the hardware architecture needed for defining a common communication protocol between software and hardware. The middleware is detailed in section 4.3. In section 4.4, an approach for software-like hardware debugging will be presented. Section 4.5 shows a case study where we use OoRCBridge in robotic application. The chapter will finally be summarized in section 4.6.

4.1 Overview

As discussed in chapter 2, there are many advantages to use FPGA with high level software in embedded or robotic applications. However, the interfacing problem between FPGA and high level software always remains problematic. Especially, when one want to integrate FPGAs in existing SW system which results a hybrid heterogenous system (different devices, physical interfaces, or high-level SW systems, etc.). *Late binding* approaches such as *language-based* approaches (HLS) allows using a unique model for SW/HW co-design and abstracting SW/HW communication. However, these approaches suffer from a low scalability since they are heavily architecture-dependent (compiler, interface, etc.). Therefore, they are not versatile enough to adapt to different devices, interface or software systems. We argue that a solution based on a combination of *early binding* approach and *platform-based* approach support better system modularity. By focusing on the separation of concerns and the interface standardization, these approaches promote reusability and adaptability, thus enhance the system scalability. Basically, the proposed approach must have the ability to easily adapt to different kinds of physical FPGA-processor communication interface. This may be a general purpose interface (USB, ethernet, serial, etc.) or a dedicated high performance SoC interface (WIEM, PCLe, etc.). Moreover, the solution needs to be as generic as possible to maximize the reuse of the system on different application contexts. This can be done by defining a uniform middleware for SW/HW communication. Beside being flexible, this middleware must provide an intuitive protocol to high level software. The protocol has to be software-friendly and requires less effort on hardware processing from software.

Traditional software languages are designed to work efficiently with memory access. Therefore, a natural way to access the underlying resources from software is to map each hardware device to a segment of virtual memory. In doing so, all of the actual I/O interaction on hardware now occurs in memory in the form of standard memory addressing. This can be achieved by using memory mapped file technique as in operating systems such as Unix. Each hardware device appears as a device file in the system. This file can be easily mapped to a segment of virtual memory. Registers on hardware devices are associated with address values. The software can therefore treat the device as if it is a part of the primary memory. The same method can be applied on FPGA by mapping the FPGA into a virtual memory region where each FPGA circuit occupies a segment of it as shown in figure 4.1.

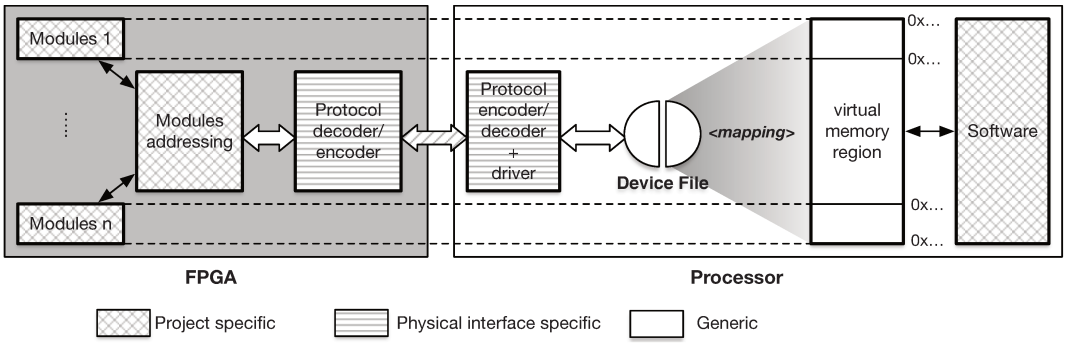


Figure 4.1: Memory mapping is used to provide a convenient access to FPGA from software. The solution must be as generic as possible to enable the reuse of the system.

A flexible FPGA/SW communication require an agreed handshake protocol. On the software side, this protocol is represented by an hardware driver. On the one hand, the driver provides a device file that supports the memory mapping technique. On the other hand, it encodes memory access requests from high level software to a format transferable via the physical interface. On the FPGA side, the reverse process is needed to decode the incoming data to address/data IO requests. Since there are maybe many independent modules (user circuits) running on the FPGA, a *modules addressing* step is necessary to provide access to the right module based on the input address (from software). The returned data from FPGA can be transferred to software using the same protocol.

An advantage of this approach is that the high level software is completely separated from the underlying system. Software just needs to access the virtual memory region in a convenient way. Nevertheless, at middleware and hardware level, there are two major obstacles that prevent the generalization of the solution. Firstly, the *protocol encoder/decoder*, both on software and hardware (FPGA), are physical interface specific. They vary depending on different kinds of interface between FPGA and processor. This is obviously unavoidable. However, the change can be made once, and then can be reused on different projects using the same interface configuration without any problem. A library of drivers/hardware wrappers can be built to support frequently used configurations. Secondly, the *modules addressing process* is project specific. Depending on applications, the modules used on FPGA maybe different. Therefore, the registers addressing mechanism differs from project to project. This problem can be overcome using an automatic code-generation approach. Based on the input modules, the registers addressing scheme can be accordingly generated. This can be done at the *post-design* stage of FPGA circuits before being synthesized on actual hardware.

The solution to the presented obstacles are addressed in OoRCBridge, our dedicated toolset and middleware for integrating FPGA to high level software. With the help of the OoRC meta-model, the approach relies strongly on an automatic code-generation mechanism. Basically, the integration process is performed in two stages:

1. *Design stage*: at this stage, the input designs, both regular designs or those imported from legacy source, are analyzed by the meta-model. This allows to collect all registers needed to be addressed for accessing from software. An addressing scheme is then generated automatically from these registers. The *encoder/decoder* modules of the supported physical interface are also assembled in this process. The stage leads to a final design that encapsulates all input modules and the protocol necessary for communicating with software.

2. *Deployment stage*: the bitstream generated from previous stage is deployed on the FPGA. On the software side, the corresponding hardware driver is installed. The system then generates all necessary APIs to allow accessing to FPGA registers via our middleware. This middleware is application independent since it is based on memory mapping technique.

Accessing FPGA circuit registers by address requires an address resolution mechanism. An address/data IO interface is suitable for this purpose. The implementation of OoRCBridge uses Wishbone interface as base addressing scheme for FPGA modules. Wishbone is an open source address/data IO logic bus* intended to modules integration. This interface† is used widely on FPGAs to connect several designs together. On the software side, the OoRCBridge is optimized for software development using Pharo (Smalltalk). However, all (software/hardware) principles presented in this thesis are generic and can be applied on other hardware buses (e.g. AXI 4.) or object oriented programming languages (e.g. Python).

4.2 Hardware Architecture

4.2.1 Interface Template

The automatic encapsulation of inputs modules within Wishbone interface happens at the *design stage*. OoRCBridge has a generic interface template that allows to automatically embed any input designs. This template is based on the integration feature of the meta-model presented in section 3.5.4, as shown in the upper part of the figure 4.2.

The template has three main parts: *WishboneMaster*, *WBBusController* and *DynamicWishboneSlave*. The *WishboneMaster* is an architectural abstract design. It defines only the regular master interface to the bus controller. A subclass of this design has the responsibility to implement the detail communication (decoder/encoder) between Wishbone and a specific physical interface. The *WEIM2Wishbone* is an example, it describes how an incoming data from the physical WIEM (Wireless External Interface Module) can be adapted to Wishbone and vice-versa. Each realization of *WishboneMaster* is specific to a physical link. One can easily develop a library of classes describing different Wishbone adapters for most popular used physical interfaces. As long as a new master is implemented based on *WishboneMaster*, it can be used directly by the Wishbone template without any further modification.

A *DynamicWishboneSlave* is a composite design of a user-input design. It encapsulates automatically the input design in a Wishbone slave interface. This can be done by connecting all logical ports (specified by S_i, *ports classification*) of the input design to a corresponding registers. These registers are Wishbone data words aligned (8/16/32 bits aligned). They are then assigned automatically to a virtual address. Therefore, they can be accessed from the slave using the address bus. All other ports classified as physical will be forwarded directly to the bus controller for external communication.

A *WBBusController* is a composite design of a master and several slaves. Internally, it defines an arbiter that decides which slave is activated at a time based on the requested address from the master.

*Wikipedia: A logic bus does not specify electrical information or the bus topology. Instead, the specification is written in terms of "signals", clock cycles, and high and low levels. https://en.wikipedia.org/wiki/Wishbone_(computer_bus)

†It is defined to have 8, 16,32 and 64 bits buses. In our system, only 8,16 and 32 bits buses are supported (which can be easily converted to computer type on software side)

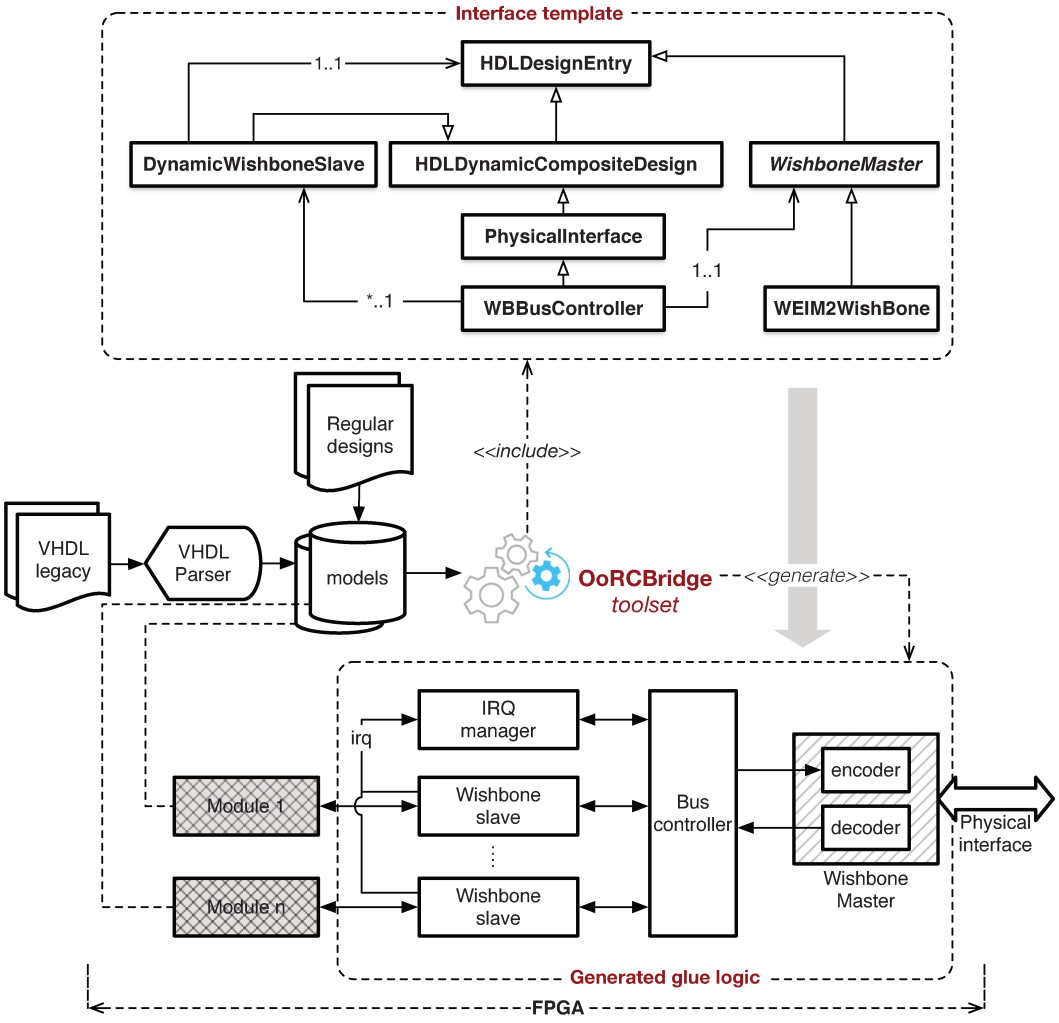


Figure 4.2: In OoRCBridge, OOD is used to design interface template. Designs are reused , refined and enriched to provide a generic, automatic and modularized IPs integration mechanism.

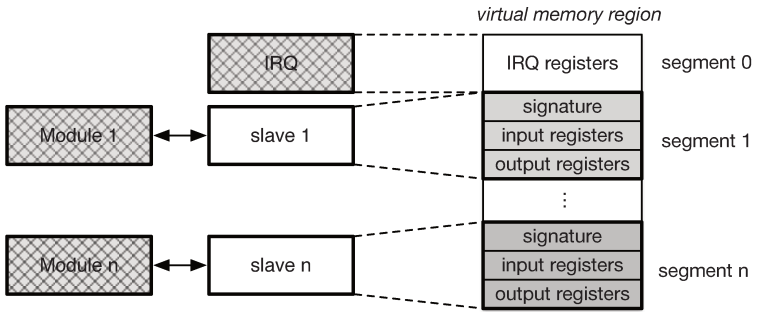


Figure 4.3: Addressing scheme generated automatically by the bus controller.

4.2.2 Addressing Scheme

OoRCBridge conserves a fix virtual memory region for mapping an FPGA. The size of this region is configurable[‡]. All FPGA registers need to be addressed in this region. This latter is then divided into several segments with variable sizes by the *WBBusController*, as shown in figure 4.3.

The first segment is dedicated to a built-in interruption manager (IRQM). This IRQM is a specific kind of wishbone slave. It connects to other slaves using an *irq* signal. In this way, other slaves can inform the IRQM whether there is an interruption (e.g. when target circuit finishes the calculation or a hardware breakpoint meets its condition, section 4.4). The IRQM has some special registers that are addressed for accessing from software: (1) pending register for identifying the pending interruptions needed to be handled; (2) the mask register allows software to enable the IRQ on a specific slave; (3) the ACK register, by writing to this register, the software can inform the IRQ manager that an IRQ has been handled. Moreover, the IRQM issues also a global IRQ signal which can be connected directly to processor to achieve an hardware interruption or can be accessed from software as a normal register (software interruption). The IRQM is generated and addressed automatically by the *WBBusController* based on used input designs.

The bus controller associates a virtual memory segment below IRQM to a slave. Each slave is assigned a base address as the beginning address of the segment. From a slave (*DynamicWishboneSlave*), each register has an offset related to this base address. The first offset holds the signature number for circuit identification. It is followed by input registers then output registers mapping. The slave can easily identify which register will be accessed based on the input address from the bus controller. Note that a long register (more than one word) is accessed by multiple consecutive addresses. From software perspective, this can be considered as accessing array elements. A slave will be activated for accessing by the bus controller when the input address from the master falls into its memory segment.

4.2.3 IPs Integration Supporting Memory Mapping

Figure 4.2 shows a complete process of user IPs integration using the proposed interface template. OoRCBridge provides a generic mechanism that automatically embeds any IPs –both designed by our meta-model or imported from VHDL code– in an interface supporting memory mapping. Thanks to OOD, the solution is modularized. It is *project independent*, since User's IPs can be designed separately

[‡]By default it is allocated 64KB

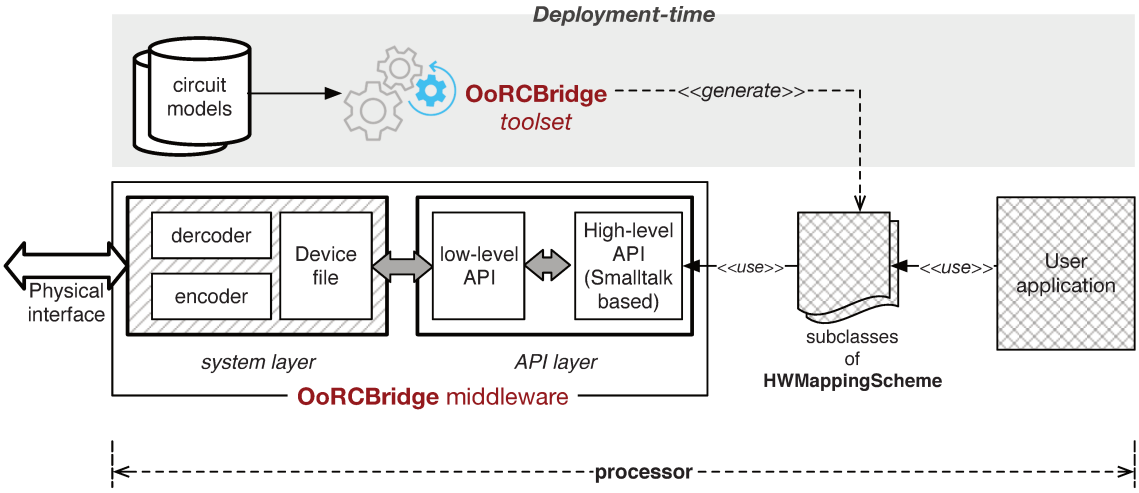


Figure 4.4: OoRCBridge middleware provides generic APIs for hardware communication. The OoRC toolset generates automatically application-specific accessing classes using these APIs

and easily plugged to the interface without worrying about IPs compatibility. It allows also physical interface abstraction, since it provides a generic adapter template for connecting (back and forth) a specific physical link to a wishbone master.

4.3 Middleware for SW/HW Communication

The presented hardware architecture is accompanied with a dedicated middleware supporting the same memory mapping mechanism. The goal is to provide a productive software environment for hardware accessing. For that, the middleware is designed with ease to use and portability in mind. By ease to use, we mean software friendly and effortless. The middleware is portable since it is independent from the meta-model and OoRCBridge toolset, thus can be used individually. It has also small footprint and hence is resource friendly. Chapter 5 will show a use case where the middleware is extended and used on an actual embedded device. Furthermore,since it is generic, the middleware is easy to integrate to various system. For example, it can be used as a ROS client (Robotic Operating System[§]) in a robotic system and provides access to the sensor on FPGA, as shown in 4.5.

Figure 4.4 shows an overview of our OoRCBridge middleware. Basically the middleware consists of two independent layers: *the system layer* and the *API layer*.

4.3.1 System Layer

This layer is the user space I/O driver dedicated to the physical interface. On the first hand, this driver handles the communication between FPGA and the processor. On the other hand, it provides a generic device special file (under /dev on Linux like systems) to the high-level software. This file can be mapped to a virtual memory region. Similarly to the Wishbone bus on FPGA, this driver is hardware specific and must be updated accordingly when we adopt a new physical interface. However, to be used by the higher layer of the software framework, all drivers must respect some constraints: (1) they need

[§]http://ros.org

to provide the same memory mapping mechanism to the high-level software, since the higher layer operates independently of the lower one; (2) they must be able to encode the address/data from higher layer to a data format compatible with the physical interface (serial, parallel, etc.); and (3) they share the same communication protocol with the corresponding Wishbone master on FPGA, so that, the master can decode data it receives from the driver and map it back to the wishbone.

One can have a library of pre-built drivers for commonly used physical interface, likewise the Wishbone master on hardware side. These drivers can be easily added into the middleware without any further modification at API level. Different drivers can be activated at the same time to provide access to different FPGA devices.

4.3.2 API Layer

The *API layer* is completely decoupled from the *system layer*. The only link between them is via the device file (produced by the *system layer*) which is configurable. The layer has two API levels. The first one is the *low-level API* which is developed in C. At this level, the system uses the memory mapping technique on the provided device file. The whole FPGA is considered as a virtual memory region. Every FPGA register can therefore be accessible via its corresponding virtual address. Furthermore, the API is able to access the IRQ Manager on the FPGA and makes the interruption handle available to users. This feature allows user applications to react to an event raised on the FPGA, for example, when the circuit finishes the processing or when a hardware breakpoint meets its condition, etc. The second level –*the high-level API*– is a high level language binding of the low-level API. OoRCBridge middleware has a Smalltalk implementation as an example of binding. The same principle can be used to bind the API to any high level object-oriented languages (Python, Ruby, etc.). User application, which use the *high-level API*, can therefore interact with FPGA registers as if they were plain objects.

In our system, different FPGA devices can be used simultaneously. Each one is mapped to a separated virtual memory region. Multiple circuits can run in parallel on one or more FPGAs. They are assigned to independent memory segments. On software, these circuits can be considered as processes. Since Smalltalk supports concurrency at the language level, it's trivial to map each circuit on FPGA to a equivalent Smalltalk process. Note that, on hardware, the circuits are independent and there is no physical connection between them, so the synchronisation between processes must be done on the software side. This can be achieved by using the interruption handle feature to determine which process is finished and is ready for synchronizing.

4.3.3 Software Development Using the Middleware

Both API levels can be used for software development, but the *high-level API* is more flexible and software-friendly since it is coupled with an object oriented language. At this level, more abstraction can be added to hide all hardware aspects. Hardware accessing can therefore be performed in a convenient way as plain software objects. In OoRCBridge, this is achieved using an automatic code generation process as illustrated in figure 4.4. When circuit models are deployed on hardware, based on the addressing scheme performed at the integration phase, the OoRCBridge toolset will generate automatically an accessing class for each circuit. This class –subclass of *HWMappingScheme*– en-

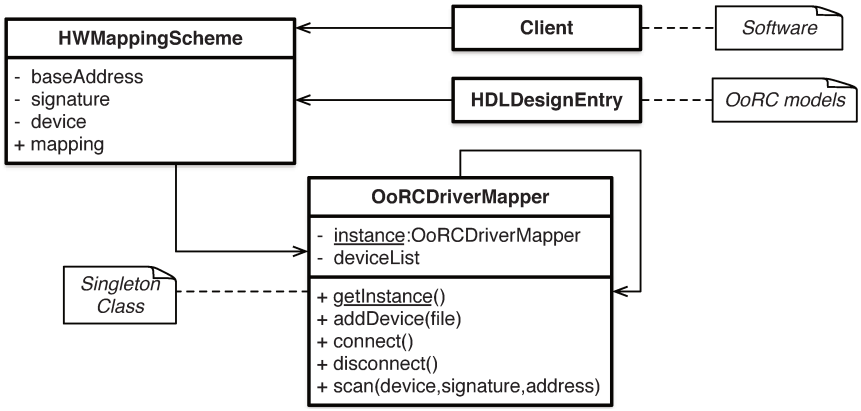


Figure 4.5: *HWMappingScheme* abstracts and encapsulates hardware circuits as regular software objects. It can be considered as a gateway for hardware accessing from software

capsulates all informations needed for identifying a circuit, such as signature, base address, register addresses, target device, etc., as shown in figure 4.5. User software can use these classes as simple *data classes* in which attributes represent registers and are accessed via accessor methods. Underlying hardware communication is handled automatically with the help of a singleton driver mapper object, an instance of class *OoRDriverMapper*. This class –a part of the *high-level API*– serves as device manager and monitoring.

Our toolset currently generates Smalltalk classes, but the feature can be easily extended to support other binding of the *high-level API*. This opens the potential use of the middleware and toolset on various systems.

4.3.4 Impact of the Middleware on the Performance of the Link

The downside of a generic middleware is that it introduces obviously overhead and sometime redundancy to the SW/HW communication. This is because of the additional hardware interface on the hardware and different software layers as well as API levels on software. To study this impact, we have performed an read/write test of the middleware on an actual device. We have used the APF51 Single Board Computer [Arm] which adopts a Freescale i.MX515 (Cortex-A8 @ 800MHz) running on 512MB of DDR RAM and a Xilinx Spartan 6 (LX9). The middleware runs on the ARM which communicates with the FPGA via a WIEM interface.

A Single Clock Block RAM design (in VHDL) was taken as the test circuit. This block RAM has 2KB of capacity (1024x16 bit) and allows a read/write operation at single clock. Based on this design, we have performed the read/write test on three distinct cases:

1. *Without the generated interface and middleware*: the interface between processor and FPGA is application specific. It has been created manually by modifying the VHDL code and has been optimized for efficiently working with the Block RAM. Concretely, the data/address bus of the WIEM was multiplexed directly to the data/address registers of the block RAM. So the data comes from the WIEM goes directly to the block RAM and vice versa. This is the ideal case since it minimizes the delay between the interface and the block RAM. The read/write operation has been controlled by a dedicated low level software running on processor.

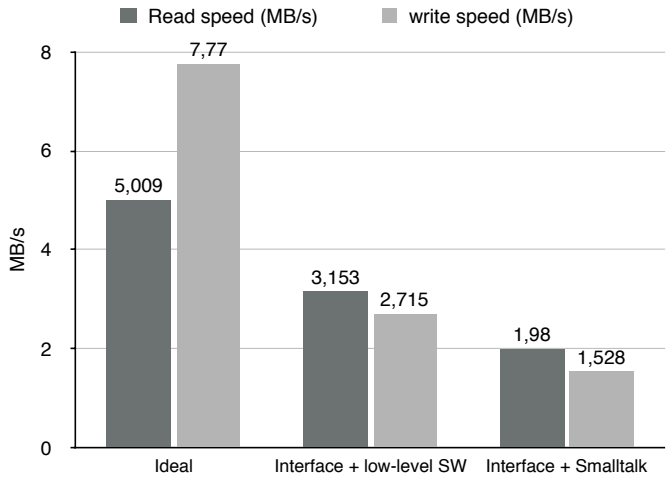


Figure 4.6: Performance measurement for continuously read/write test

2. *With the generated interface + low-level software API*: the Block RAM design has been imported to our framework and automatically integrated with the Wishbone interface without any further modification. Therefore, the data/address registers of the block RAM were accessed via the wishbone interface by using their associated address. On the software side, a small C application using our low-level software API is developed to provide access to these registers.
3. *With the generated interface + high-level software API*: This case uses the same hardware configuration as the second one. But on the software side, the toolset generates also the accessing classes. The application has been developed in Smalltalk and used these classes to perform the read/write operation to Block RAM registers on FPGA.

For the last two test cases, the wishbone has been configured conforming to the WIEM on AFP51 with 16 bits of data width, 16 bits of address width and 100 Mhz clock.

The same test scenario has been carried out for these three cases: a 20 MB of data was written continuously to the block RAM (2KB) and then was read back and stored into an array. This process has been repeated 10 times for each case and the average read/write speed has been calculated. The result is shown in the figure 4.6.

In the first case, since the communication interface is optimized only for the block RAM on FPGA, an ideal transfer rate of 5MB/s for reading and 7.77MB/s for writing is obtained. This, however, takes an important amount of development time both on hardware and software which can be quantified by the number of active lines of code. The process – which performs simple read/write on the block RAM– took totally about 320 lines of code (180 lines of VHDL for the interface[¶] + 26 lines for the IO configuration + 114 lines of C code).

In the second case, to make the interface more generic, a wishbone interface was inserted on hardware side, and the accessing on software side was realized via our low-level API. Although the read-/write speed is reduced to around 3MB/s, developers can save a lot of development time since the interface was generated automatically. It took about 112 lines of codes (13 lines for IO configuration + 99 lines for the C code). They, however, must take care of direct management of data on the low-level software (registers addresses, data conversions, .etc).

[¶]Not counting the VHDL code of the Single Clock Block RAM

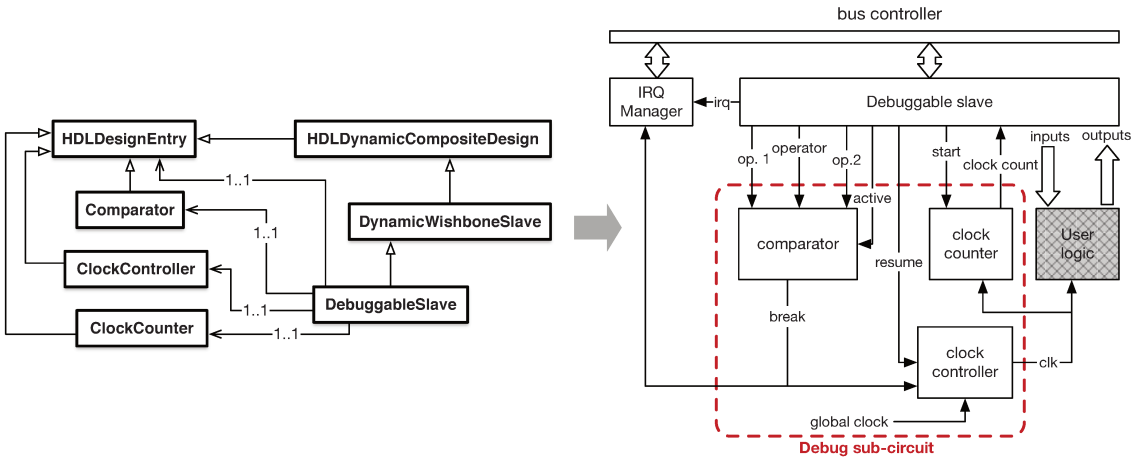


Figure 4.7: *DebuggableSlave* allows to inject automatically a debug sub-circuit to the slave and turns it to a Breakpoint controller

The last test provides a complete object oriented software solution with the generated classes. The API layer takes care of the address mapping and data conversion. Obviously, there is a drop in the speed to about 1.8MB/s, since this introduces yet another software layer. However, the process took only 43 lines of code (13 lines of IO configuration + 30 lines Smalltalk). There is a trade-off between run-time performance and a faster development and eventually a shorter time to market, thanks to the use of software, and to higher-level abstractions. Our proposed solution let developers decide where to put the cursor.

4.4 Hardware Controllability and Debugging

A key issue is to allow high-level applications to easily interact with the hardware part on FPGA in a software-like manner. An efficient interaction raises the question of how to use the debugging software technique on hardware. To enable software-like debug capabilities on hardware, we adopt our concept of dynamic hardware breakpoint presented in [LLF+15] into the framework. The idea is that, when the meta-model generates the slave for a target circuit, if the debug feature is enabled, a debug specific sub-circuit and breakpoint control logic are injected automatically into the slave. Figure 4.7 depicts the design and the generated sub-circuit.

To enable debug, the *DebuggableSave* design is used to encapsulates a target model instead of *DynamicWishboneSlave*. Beside connecting and addressing circuit registers, the debuggable slave also acts as a breakpoint controller. It has three additional components. The first one is the breakpoint *comparator* which takes three inputs from the slave. These inputs consist of two operands and an operator (=,<,>,/=). One operand can be set manually by software. It stores the reference value for comparison. The other operand is one of circuit's output registers selected automatically by the control logics. The selection is based on the address of the register configured as breakpoint from software. When the breakpoint is activated and the condition on the *comparator* becomes true, the *comparator* triggers the *break* signal which connects to the second component, the *clock controller*. This component gets the global clock as input and issues a controllable clock signal to the target circuit. This controller can

disable the output clock when a control signal is triggered (i.e. the *break* signal). Cutting off the clock results in stopping the execution of the target circuit while holding its current state. Since the slave uses the global clock, it is not affected by this event and thus can read the target circuit's state on software demand. The *break* signal is also used by the slave to trigger an interrupt to the IRQ Manager. This allows the manual handle on software side when the breakpoint takes place. In addition to the *clock controller*, a third component, the *clock counter*, is added to measure the execution time of the target circuit in clock cycles. It starts as the circuit to debug starts the computations and stops when the done signal of the circuit is asserted. Since the counter uses the same clock as the circuit, it is also halted when the clock is disabled. In this way, the developer can inspect exactly the execution time of the circuit at the breakpoint.

```
1  cnt := HWCounterMapping new.
2  cnt input:100.
3  cnt setBreakpointOn:#output forValue:50 condition:#=.
4  cnt start:true.
5  cnt waitForIRQ:[
6    cnt bpActive ifTrue:[
7      ('Stop at: ', cnt output asString) print.
8      ('Steps: ', cnt clockCount asString) print.
9      cnt resume:true.
10   ] ifFalse:[
11     'Execution done' print.
12   ]
13 ] timeOut:10 milliseconds.
```

Listing 4.1: Example of using hardware breakpoint in software. The *HWCounterMapping* is the accessing class of a simple hardware counter. This counter has an *input* and an *output* signal, and counts from 0 to the value of *input* (100). The breakpoint is set for *output* at value 50 (first operand). Note that the slave uses the address of *ouput* to select the second operand for the comparator

Software can resume the halted execution at anytime by writing a true boolean to the *resume* register which connects to the *clock controller*. This action allows the *clock controller* to enable the output clock, and hence wake up the target circuit and the clock counter. The execution flow can then continue. This debug feature is built in supported by our low-level API and is easily bound to any high level binding (e.g. Smalltalk implementation).

The listing 4.1 shows an example (in Smalltalk) of how the breakpoint is set on the software side and how to use the IRQ Manager to handle the breakpoint. *HWCounterMapping* is a subclass of *HWMappingScheme*. This class –generated by the toolset– abstracts a simple hardware counter circuit on FPGA. The program simply waits until interrupt happens then prints the value of *output* and the number of executed clocks at the breakpoint. The execution of the circuit is resumed (line 9) after the breakpoint is processed.

The only drawback of this method is that the implementation of the debug circuit requires to use the clock-gating technique to control the clock (*clock controller*). The vendor specific Digital Clock Manager (DCM) is required to produce a low-skew gated clock. This feature, therefore, is vendor dependent. Vendor specific features are limited in our system since we focus on a hardware/software independent platform to enforce the portability between systems.

4.5 Case Study: Using OoRCBridge Toolset and Middleware for Robotic Development

Today robotic computing systems are usually implemented using general purpose processors because of their accessibility and simplicity which do not require specific knowledge. Furthermore, many robotic middleware are available that facilitate the development process. However, this approach restricts several optimization opportunities and may not always satisfy performance, cost, and energy requirements [CWFH13]. FPGA infrastructures can be considered as a good solution for these issues, especially in complex robotic systems that require time consuming tasks. Advantages are many to use FPGAs along with general purpose processors. On the one hand, FPGAs provide hardware acceleration and on the other hand, CPUs allow developers to use flexible software development environments. This combination also allows reduce the overall energy consuming of the application by carrying critical tasks on FPGA and hence deducing the software overhead.

However, software/hardware integration remains a challenge for robotic developers and usually results in a loss of productivity [BRS13]. Robotics development involves experts from different domains. To encourage them to adopt FPGAs in their projects, a unified software/hardware platform for easily integrating FPGAs in existing robotic system is mandatory. Above all, this platform must be generic enough to be reused from project to project with minimal modification. OoRCBridge is well suited for this purpose. Firstly, it eases the software/hardware integration with the help of a dedicated middleware and an automatic code generation process. Secondly, its API can be easily integrated to other robotic middleware – by binding the low-level APIs to the new environment. This allows developers to stay on their robotic middleware while have additional hardware accessing feature.

This section demonstrates a use case of OoRCBridge in robotic development where a robot uses a camera to detect and follow an object specific by a colour pattern.

4.5.1 Scenario

The robot follower is developed using the ROS middleware. The application consists of a ROS network with many nodes for controllers and sensors. We use Pharo Smalltalk for the implementation of ROS nodes –as the Smalltalk binding of ROS client API is available.

Among the sensor nodes, there is a node that handles a camera –called *detector node*. It is the principal node for object tracking. The principle is simple, the node will capture images from the camera and use a colour filter algorithm to detect object position. This latter will then be communicated to the controller node via the ROS network. Since a software implementation of the image processing can easily be achieved, it is not an optimal solution. The application require a real-time tracking from the *detector node* which is therefore a time-critical task. A software implementation may not fast enough to respond to this characteristic, especially for a large image resolution (e.g. VGA). Furthermore, the use of a high level language to implement the task can add more overhead to the system and thus can increase the overall energy consummation –another factor that should be considered in such robotic application. The comparison in section 4.5.3 will verify these hypothesis.

FPGAs are suitable for resolving these kinds of problem. In this demonstration, we focus mainly on the implementation of the image processing algorithm –critical part of the node– on FPGA. For simplicity, we use the same device (i.e. APF51) as the previous experiment to build the detector node. The idea is

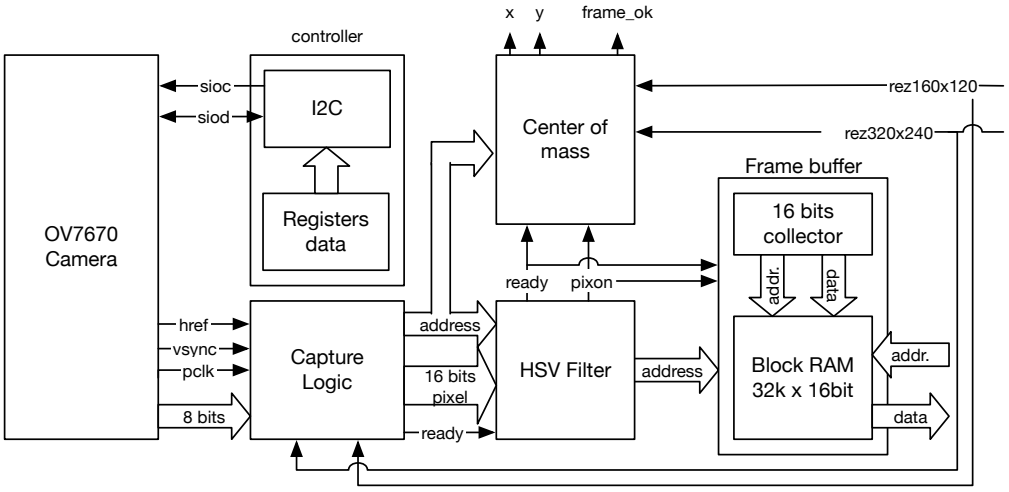


Figure 4.8: Original design of the detection circuit

that we connect directly a 0V7670 camera to the FPGA (APF51) via its GPIOs. The FPGA captures the image from the camera (pixel by pixel) and filters each pixel using a hardware HSV filter by a specific colour pattern. The filtered pixels are then used to calculate the barycenter of the detected region which finally provides the position of object. The hardware design of this process can be described using our meta-model, but here we use existing VHDL design from another project instead. This allows to demonstrate the ability of reusing existing third-party designs (backward compatible) of the meta-model.

Figure 4.8 shows the original design of the detection circuit, it contains 5 main components:

- *The camera controller* is used to configure the functionalities of the camera via a I2C-like interface. The two important configurations used in this experiment are: (1) the pixel is 16 bits RGB 565 format and (2) the image is in VGA mode (640x480).
- *The capture logic* controls the image acquisition. From here the resolution of the image can be configured by either 640x480 (VGA) or 320x240 (QVGA) or 160x120 (QQVGA) via dedicated flags. This is achieved by downsampling the input image from camera by a factor (e.g. for the QVGA mode, for every 2 lines 1 is skipped and every 2 pixels, only 1 is captured).
- *The HSV filter unit* filters each pixel by converting it from RGB to HSV format and then test if it falls into the threshold colour range of a specific colour pattern. This unit outputs a 1 bit binary pixel.
- *The center of mass unit* uses the binary pixel from the HSV filter and the pixel position to calculate the barycenter of the detected region. The unit assumes that there is only one region in the filtered image. The object position will be the barycenter of all regions.
- *The frame buffer* stores the filtered image into a block RAM, each pixel is encoded as 1 bit (binary image).

This design has been imported to our system using the meta-model without any modifications. Each component corresponds to a design class. The OoRCBridge toolset has been used to generate the

The HSV colour space is used because it is less sensitive to lighting variations

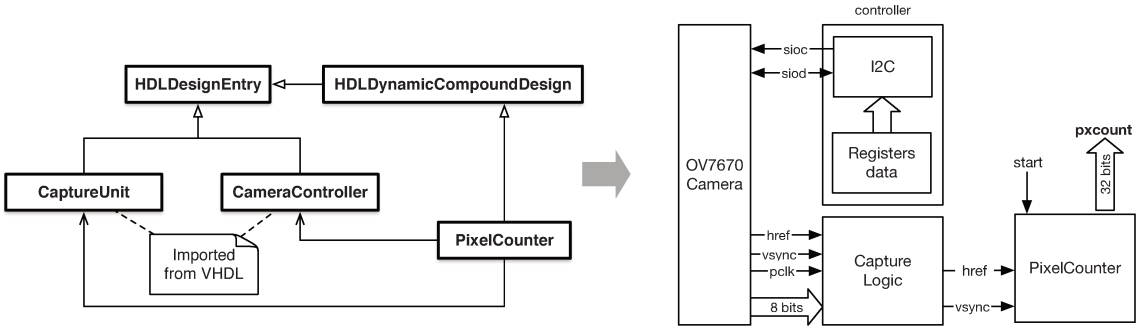


Figure 4.9: Mixture use of imported VHDL designs and custom design using OoRCSCript. The pixel counter simply count all received pixels

communication interface as well as the necessary accessing classes for the middleware. The only task that remains to be done manually is to create a configuration class that provides the IO pins specification (subclass of *DeviceConfiguration*). This class is only about 25 lines of code. The final design model (including interface) has been then exported to VHDL to perform a low level synthesis using vendor tool. This has been done automatically by using our toolset.

With the help of the generated accessing class, it becomes trivial to configure the image resolution on the capture unit from OoRCBridge middleware using Pharo. The IRQ manager is used to inform software whether an image is completely processed and the object position can be read. These classes also allow access to the filtered image (on the frame buffer unit).

4.5.2 Debugging Using Hardware BreakPoint

Despite allowing multi-resolution capturing, the original design works correctly only with the QQVGA format. We've tested it with larger image formats. In these cases, the filtered image was very noisy and the detection showed incorrect result. This is probably due to the fact that the *HSV filter* takes more processing time on a pixel than the necessary time for a new pixel to be available from the *capture logic* when capturing a line. For the QQVGA format, the *capture logic* produce 1 pixel for every 4 received pixels. This may meet the timing requirement for the filter. But when increasing the resolution, the *capture unit* takes shorter time to produce a pixel, and hence this can cause the problem.

To verify this hypothesis, we've first simulated the filter and noted that it took around 14 global clocks cycles to complete a pixel. We face a challenge when attempting to measure the pixel capturing since it cannot be simulated. The *capture unit* needs to be connected directly to the camera and uses the pixel clock provided by the camera along with the global clock. A solution was to use the hardware breakpoint for this measurement. With the help of the meta-model and OoRCScript, a simple pixel counter model was created using the *capture unit* as sub-circuit as shown in figure 4.9 (This counter simply counts the number of captured pixels in a frame). The debug option has been enabled on that model so that the toolset could inject automatically the debug sub-circuit when generating the interface for it. After deploying the pixel counter on FPGA, the software has been used to control the hardware breakpoint. Concretely, this latter has been configured to stop the execution when a line was completely captured (640 pixels, at VGA resolution). By reading the clock counter at the breakpoint,

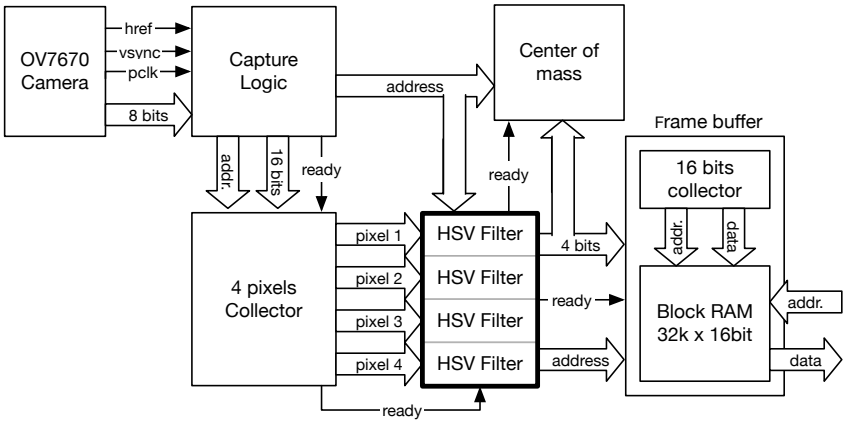


Figure 4.10: Optimized design of the detection circuit with 4 HSV filters in parallel

we've acquired a value around 2556 global clocks cycles** (for VGA setting). So that it takes about 4 global clocks cycles to produce a pixel at the highest resolution (shortest pixel production time on the *capture unit*). This result confirms the proposed hypothesis.

A possible solution to this problem is to collect 4 pixels ($\sim$ 14/4) from the *capture unit* and then process them in parallel by using 4 separated filters. At highest resolution, the time of collecting 4 pixels (16 clocks cycles) is approximatively the processing time of a pixel on the filter unit (14 clocks cycles). By pipelining the *capture unit*, the *filters unit*, and the *center of mass unit*, the pixels processing part takes no extra time compared to the pixel capture time. This was done by modifying the VHDL code of the original design. The new version is shown in the figure 4.10. Since this new design is optimized for the highest image resolution (VGA), it obviously works well with the lower ones. Again, the design can be imported and used in the framework without any problem.

4.5.3 FPGA *vs* Processor

As mentioned previously, using FPGA to handle the complex processing task can improve the performance while reducing the power consumption. To verify this hypothesis, we have performed an experiment based on object detection algorithm with two scenarios:

Image processing using software: The FPGA is used to acquire image from the camera sensor and store it in a local block ram. The processing software is developed in C and runs on the ARM. This program continuously fetches image from FPGA's block RAM and filter it (in HSV colour space) based on a colour pattern. Since the internal BRAM (Block RAM) is limited, the FPGA can only store each time an image colour of QQVGA resolution (160x120). Therefore the scenario works narrowly on QQVGA image.

Image processing using FPGA: In this scenario the image processing part is handled by the FPGA. The optimized detection circuit on figure 4.10 is used. For the experimentation, the circuit is configured to work with VGA image. In this scenario, the software part simply fetches the object position as soon as it is available.

** Note that, here we measure the number of global clocks cycles, not of pixel clocks cycles that are provided by the camera

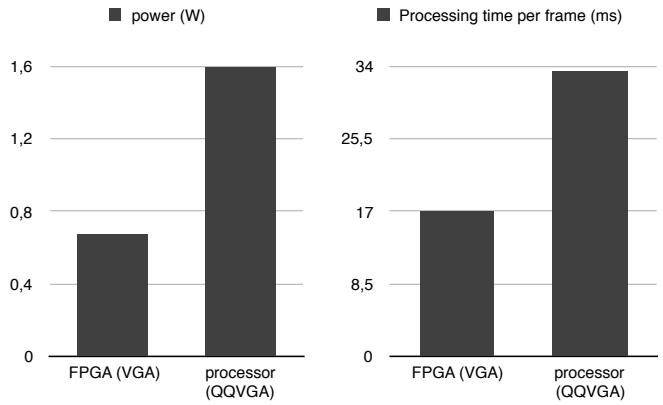


Figure 4.11: On the left, the power consumption between the software and hardware implementation of the object detection. On the right, the processing time per frame of each version

Figure 4.11 shows the average power consumption and the processing time of each frame between the two scenarios. The result demonstrates that, the VGA image processing on FPGA is two times faster than the QQVGA software version while consuming two times less power. The first scenario is performed entirely sequential. So the total processing time for each image is denoted as : $t_1 = t_c + t_{tf} + t_p$. With t_c is captured time, t_{tf} is the image transfer time and t_p is the processing time on processor. In the second scenario, since the data is streamed, the filters are in parallel and the processing units are pipelined, the processing part takes no extra time compared to the capturing part, that is: $t_2 = t_c$. This proves why the image processing on FPGA (VGA) is much faster than on the processor (QQVGA).

Regarding the power consumption, the first scenario requires energy on both FPGA (for capturing) and processor (for processing). In the second scenario, only the FPGA is in need of power, the processor only fetches the object position from the FPGA and hence, has less impact on the power consumption.

Literally, the work on [CWFH13] has proved that FPGAs outperformed GPU and CPU for fixed algorithms using streaming such as digital signal processing, or data encryption, etc.; which is exactly our case.

4.5.4 Communication Through the ROS Middleware

The detector node's software runs on the ARM processor of the APF51 board. The Smalltalk binding of the ROS client API is prior installed on the node. Since our middleware API also supports Smalltalk binding, it is straightforward to develop application using a mixture of the two middlewares without any problem, as shown in listing 4.2.

```
1 |node pub hwdetector pos|
2 "Hardware accessing class generated by the middleware"
3 hwdetector := ODMappingScheme new.
4 hwdetector resolution:4 "VGA".
5 hwdetector start:true.
6 "ROS API for communication via ROS middleware"
7 node := ROSNode new.
```

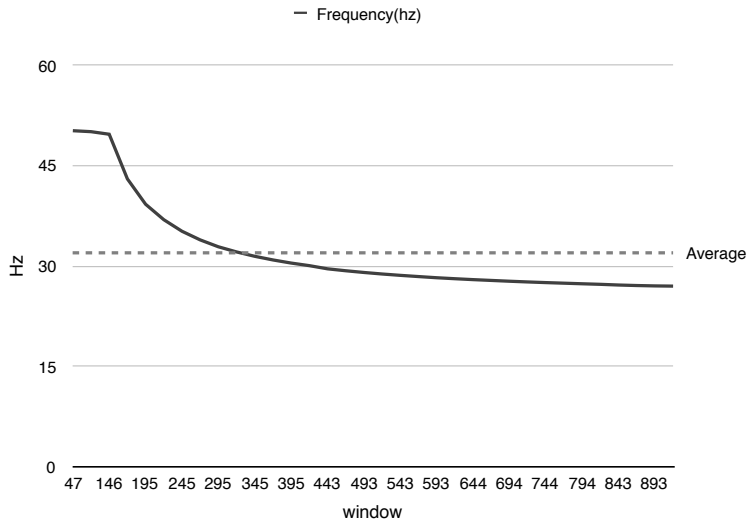


Figure 4.12: Publishing frequency of the topic */spybot/objectpos* in regarding different window sizes of messages

```
8  node name:'/detector'.
9  node master:'192.168.10.100' at:11311.
10 pub := node createPublisher:'/spybot/objectpos' type:'std_msgs/Int32
       MultiArray'.
11 node deploy.
12 "main loop for publisher"
13 [node rosOk] whileTrue:[
14   [ hwdetector fram_ok = true ] whileFalse.
15     pub publish: { hwdetector x. hwdetector y}.
16 ]
```

Listing 4.2: A mixture of ROS API and OoRCBridge middleware API. The code publishes object positions through the ROS middleware

Thanks to the automatic code generation feature of the toolset, the hardware accessing via our middleware is simple and convenient. Software developer only need about 16 lines of code to connect the node to the ROS network, access object positions from hardware and publish them to the controller. The code is quite comprehensive for roboticists who are not hardware experts. Without the toolset and middleware, the implementation would be more specific and complicate. Manual development will take an important amount of time on software/hardware communication and make the maintenance of the application more difficult, since each change on the hardware may cause a propagation change on the software –from low level to high level.

Figure 4.12 shows the publishing frequency of the detector node to the network. We've achieved a frequency of around 27 Hz. This frequency is quite good for a real-time tracking robot.

4.6 Summary

Software/hardware interfacing remains always problematic as a time-consuming, expert-requiring and error-prone task. It demands an important development effort while has less contribution to the overall application and can cause a loss of productivity. Nevertheless, by using middleware, the task

can be generalized and simplified at certain levels. OoRCBridge proposes a solution for this problem. It provides a highly abstract environment for software/hardware communication. The design of the system relies on the perspective of software programmers –who are not always hardware experts – and therefore is software-friendly. The middleware and the toolset allow to close the interfacing gap by abstracting the software/hardware communication. Automatic code generation (both on software and hardware) handles complex software/hardware co-design tasks and therefore avoid the error-prone problem. In OoRCBridge, some manual tasks have to be done at the very low level where the physical interface meets the hardware driver. This is unavoidable since they depend on how the FPGA is physically connected to the processor. However, these tasks can be considered as generic to a specific physical link and can be reused on different applications without any problem –as long as the physical interface remains the same.

The Internet will disappear. There will be so many IP addresses, so many devices, sensors, things that you are wearing, things that you are interacting with, that you won't even sense it. It will be part of your presence all the time...

Eric Schmidt, Google chairman

5

CaRDIN: A Dedicated Environment for Edge Computing on Reconfigurable Sensor Networks

Contents

This chapter mainly focuses on the problem of developing and deploying applications on edge-centric sensor networks, consisting of hybrid nodes. The network topology is, however, shadowed and unspecified. Since the network is based on internet protocol, any IP-based (Internet Protocol) network topology can be used with the middleware (CaRDIN) as an underlying networking layer (system level). We start with the discussion about the context and the general view of CaRDIN in section 5.1. Its

architecture is then detailed in the section 5.2. Section 5.3 describes the programming mechanism of edge-centric SN based on CaRDIN middleware. Before concluding the chapter, two case studies is showed to demonstrate the proposed platform as a proof of concept in section 5.4 and 5.5.

5.1 CarDIN: overview

Despite of various existing middlewares, developing and deploying end-to-end application on SN (in general) remains highly complex. There are alway the problems of *scalability* and *heterogeneity*. *Scalability* issue often relates to the handleability of a middleware in different SN scales. In a large scale SN, manual management, development and deployment of each node is not a good idea. Remote and dynamic methods are more preferable in this case. Such mechanism allows developing and reconfiguring the nodes without physically removing them from the deployment site. The *heterogeneity* problem impacts not solely the hardware architecture of a node but also the software design process. That is, nodes' hardware is often homogenous while software development is usually performed in a heterogeneous way. Most existing middlewares rely on a fix hardware architecture for building nodes and have difficulty to adapt different node's hardware architectures to the system. On the software side, application development involves programming both server and nodes. This result in a complex process since programmer needs to individually develop each nodes, then link them together on the server side. This process may not be homogenous since the development of the development of the server part and the nodes may use different languages, technics or architectures (processor vs. microcontroller). Hardware homogeneity limits the heterogeneity of the SN while SW heterogeneity makes the development process more complicated and error prone.

Existing middleware solutions tend to focus on data centralization, they operate on SN consisting simple nodes (HW and SW) with limited capabilities. These nodes simply perform data collection and delivery. A decentralized architecture such as edge-centric computing, on the other hand, pushes the data processing to the edge of the network and thus requires more capable nodes (SW and HW). Such nodes can handle more complex application for data processing and decision making. A truly distributed environment is therefore needed for such systems that allows to: (1) push the processing power of the application to the edge-centric node, (2) promote the development, management and deployment of distributed application on the network.

Up until now, the problems of HW design and HW/SW integration have been addressed. We have proposed dedicated toolset and middleware for SW/HW co-design on a hybrid FPGA/processor devices. However, to use the system for edge-computing, a dedicated distributed environment for edge-centric applications is still missing. The proposed middleware must take into account the development and deployment of not only the SW but also the HW on hybrid nodes. CARDIN is our proposition to solve this problem.

CaRDIN proposes a dedicated middleware, hardware architecture and toolset for edge computing environment based on IP-based SN. As discussed in chapter 2, IP-based SNs provide an homogeneous communication layer for connecting heterogeneous devices. The use of IP-based SN has two main benefits: (1) it solves the hardware heterogeneity problem by promoting the integration of wide-range of IP-compliant devices; (2) it provides the compatibility with existing IoT infrastructure and thus easily

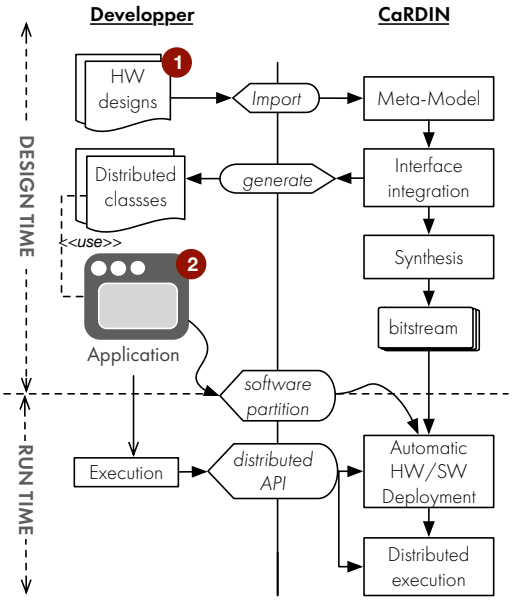


Figure 5.1: Workflow of CaRDIN. Developers need to: (1) import the HW IP to system for software/bitstream generation; (2) use the generated classes to develop their application

connects things to the internet.

In CaRDIN, to reinforce the equipment at the edge of the network, hybrid FPGA devices (i.e. FPGA + processor) are used. These devices are ideal for balancing the performance requirement with the energy consumption limitation of edge-centric nodes. They could have many form-factors and architectures. Therefore, to ease the SW/HW co-design, CaRDIN abstracts the SW/HW interaction on a node using a generic communication model (based on OORC-Bridge).

Basically, the network's architecture adopts a base station (data center, cloud) at the core where the services are exposed to the end-user. This base station is surrounded by nodes with small web server constituting a content-distributed network*. Web services (REST-based, websocket) are used as the base communication model for the network. To develop applications across the proposed network, the CaRDIN's middleware relies on a Virtual Machine solution. The VM creates a unique SW layer for application building. This provides an homogenous SW environment for the development and deployment of distributed applications on the network. The VM and web-services enable also the possibility of automatically and remotely reconfiguring SW/HW on edge-centric nodes. With these features, applications now can be developed and centralized in one place, while being executed in a distributed manner across the network.

CaRDIN consists of a predefined software/hardware architecture and a toolset that help to efficiently build and deploy edge-centric nodes. Figure 5.1 shows the workflow of CaRDIN both at design time and runtime. At design time, the base station has a toolset dedicated to interface integration. The toolset takes HW designs (could be legacy IPs (VHDL) or OoRC models) as input and generates all the required interfaces to : (1) the communication between the FPGA and the processor on the node, (2) the node-to-node and node-to-base station communication. This toolset outputs a bitstream and software API classes that will be deployed on the nodes at runtime. Developers can use the generated

*The network topology could be any IP-compliant topology

API classes along with our distributed object API to remotely access the FPGA on the nodes. From the developer perspective, the entire application is developed on one place as a regular program, while being distributively running across the network. For that, at runtime, CaRDIN provides a mechanism for automatic SW partitioning and deploying software/bitstream on the nodes from the base station.

5.2 Architecture of a Node

Since OoRCBridge is designed for SW/HW co-development, it fits well with the purpose of building edge-centric nodes. OoRCBridge can apply architectural design constraints on edge-centric nodes and allows to have a generic and homogenous application layer on top of heterogenous devices. The SW/HW interface is standardised with an uniform SW/HW communication mechanism. This allows to abstract the HW accessing from SW point of view. This separation of concerns promotes the independence of higher middleware from the underlying HW architecture. The middleware handles automatically the SW/HW communication and thus reduces the application design complexity.

Figure 5.2 shows the simple view of the proposed SW/HW architecture for an edge-centric node built on top of an FPGA coupled with a processor (e.g. ARM). We use OoRCBridge for HW (FPGA IPs) integration and HW/SW communication. CaRDIN middleware relies on APIs exposed by OoRCBridge to provide abstract hardware access to user application. Both OoRCBridge and CaRDIN are hosted by an embedded-oriented Linux OS. The IP Stack is part of the OS and is used by CaRDIN to implement the communication protocol (HTTP, REST). Our middleware is based on a lightweight (small memory footprint) HTTP server that supports plugins. These plugins can be loaded at runtime on demand. There are two core plugins: the REST engine and an embedded interpreted language Virtual Machine (e.g. Smalltalk VM).

The REST engine implements the web service mechanism that handles the network communication. The REST architecture does not have the definition of common data formats. The exchanged data can be formatted differently depending on application. XML is commonly used for data formatting, but using it on sensor nodes is not very suitable. The XML syntax is too verbose and requires a complicated parser with signification computational overhead. Javascript Object Notation (JSON) is a good alternative for data formatting. Its syntax is simple and compact. Hence, it is well suited to sensor nodes. In CaRDIN, all network messages are in JSON format.

All remote commands between nodes (in JSON) are decoded to software objects (e.g. Smalltalk objects) by the REST engine and handled by the VM. This VM has a dedicated API and primitives to access to the FPGA registers (using OoRCBridge) and to reconfigure the FPGA given the bitstream. Returned data objects from the VM can be serialized to JSON message which can be transferred to other nodes using the REST engine.

The VM along with the REST web services offers two benefits : first, since the node supports dynamic language, the software on the node can be evolved at runtime; second, the node can be reconfigured (software/hardware) remotely without the needing to restart the node. The system enables the distributed programming on the node. In other words, the software development can be centralised on the base station while its deployment and execution are actually automatically distributed across nodes.

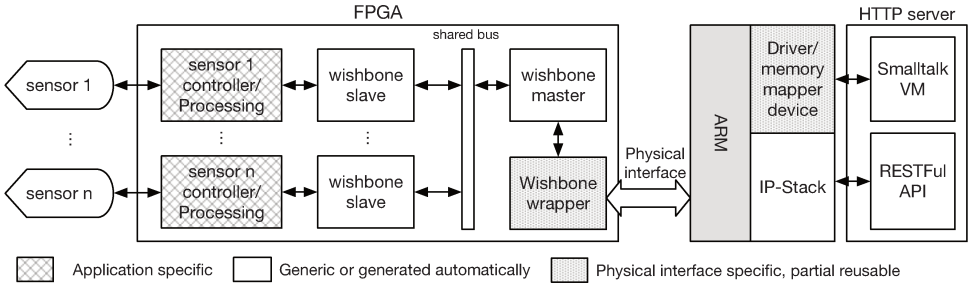


Figure 5.2: Simplified hardware/software architecture of a edge-centric node: the system is made as generic as possible by maximizing the reusability of software/hardware components

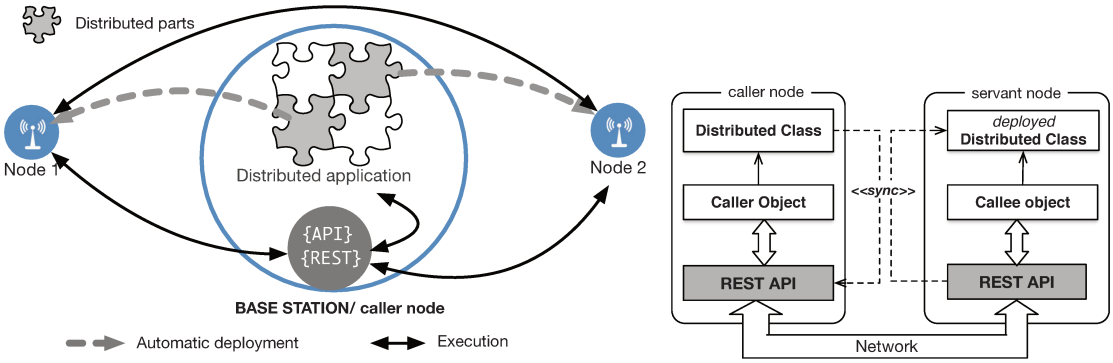


Figure 5.3: (a) The entire application is developed on base station but is executed in distributed manner; (b) Communication between distributed objects residing in the caller node and the servant node

5.3 Edge-centric Nodes Development with CaRDIN's middleware

5.3.1 CaRDIN's Distributed Object API

Distributed computing allows collaboration between objects across different address spaces of the network. These objects work together, share data and invoke (remote) methods. Regular Distributed Object APIs (DOAs) [Ora12, CS03] require the server and client parts of the distributed program to be developed separately and deployed manually. CaRDIN offers a more dynamic distributed programming environment, with the ability to deploy software at runtime, as shown in the figure 5.3a. The idea is to create an homogenous software environment where the entire distributed application can be developed in one place. Then, the system automatically handles the partition and deployment of the remote parts to the nodes at run-time. To do that, the API allows remote and local methods to coexist in a same class. In our DOA, we use the term **distributed class** for this kind of class. The system treats the methods of this class as regular methods that can be used anywhere in the program[†]. The API handles automatically the deployment of these classes on the remote nodes at runtime. They can be changed at anytime. Beside, the remote nodes are updated transparently if needed.

The proposed software API follows the principle of distributed objects architecture, as shown in figure 5.3b. Remote and local methods are written in a single class (distributed class). They distinguish from another through method annotation. The distributed classes are subclasses of a special class named

[†]A local method can use any remote method inside its body and *vice versa*

SSSynchronisableObject. This class handles the network communication between objects (caller object and callee object) using the REST API. The key different of our DOA in comparison to other DOA is that, at deployment time, the same distributed class will be deployed on both caller node and servant node. No partition of methods needed. An instance of this class can play the role of either caller (master) object or callee (slave) object. Depending on the role of this instance (caller or callee), its behaviour when executing a method is different. Concretely, if the instance is a caller object, all annotated methods (e.g. with pragma) will be considered as remote methods and will be remotely invoked using our DOA. Otherwise, if the instance is a callee object, it will recognize all annotated methods as local methods and others methods as remote methods. This mechanism allows a full bidirectional communication between objects using the same connection (for each remote call). That is, beside performing any remote call from the caller object, the callee object can also perform a remote call (e.g. callback for returned data) to the caller object on the same connection. The mechanism is applied on any node on the network, and allows a peer-to-peer communication between nodes without passing to the base station. Note that, to record changes on a distributed class, CaRDIN will assign a (new) version number to it when it is created or modified. A deployment process on remote nodes will be automatically triggered when this number is changed.

```
1  "Simple distributed class with a remote method that calculates the
       factorial value of a given number"
2  ExampleApp >> factorialOf: aNumber
3    <#remote>
4    |f|
5    "implementation of the method on callee"
6    f := (1 to: aNumber) inject: 1 into: [:product :each | product * each].
7    "remote callback to caller"
8    self printFactorial:f
9
10 "This is a normal method"
11 ExampleApp >> printFactorial: aNumber
12   Transcript show:aNumber
13
14 "This distributed class can be used by creating an object and binding it to
       the address of a node, the callee object will be automatically created
       on that node"
15 obj := ExampleApp bindTo: '192.168.1.10:9191'.
16 obj factorialOf: 10. "3628800"
```

Listing 5.1: *ExampleApp* –a subclass of *SSSynchronisableObject*– is a distributed class with one annotated method (*#factorialOf:*). On the base station, at the first object instantiation of the class (line 15), the class is automatically deployed on remote node. Line 16 requires the node to calculate the factorial of 10, then prints it on the base station

When a caller wants to perform remote call on the callee object, it initiates the communication with the remote object. The caller arguments are then serialized to JSON and passed to the callee object via the REST API. On the servant node, the REST engine receives the JSON data and reconstructs the argument objects, the corresponding method is then called on the callee object. The result of the call is finally serialized to JSON and send back to the caller object.

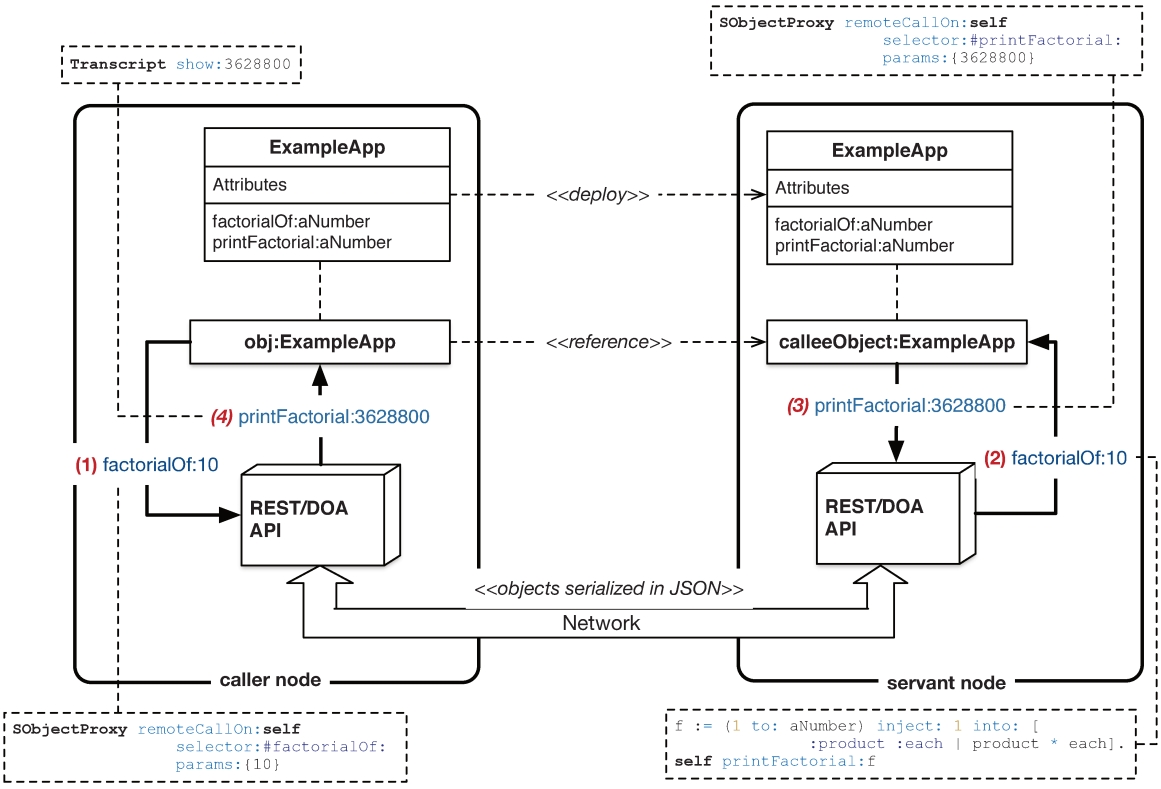


Figure 5.4: The automatic deployment and remote call of the example in listing 5.1

Figure 5.4 illustrates the execution of the code in listing 5.1 which captures the general idea of our DOA with a simple example of a distributed class implemented in Smalltalk. The method *#factorialOf:* on the caller object is recognized as a remote method (via the pragma *<remote>*), when executed, its operation is replaced by a remote call, while the actual operation (factorial) is automatically performed on the corresponding callee object. On the servant node, when the callee object invokes the method *#printFactorial:* (inside the *#factorialOf:* method), since this method is not annotated, the callee object considers it as a remote method, thus, it perform a callback to the caller object to print the factorial result on the caller node. Note that, the caller object and the callee object share the same attribute values (e.g. instance variables). These values are synchronized between nodes after a remote call.

The generated classes from the toolset of OoRCBridge are decorated using subclasses of *SSSynchronisableObject*, that enable the remote FPGA accessing. They provide all necessary API methods to remotely access the corresponding FPGA circuit registers on the sensor node. These classes can be easily extended by manually adding more methods or by being subclassed.

5.3.2 Automatic Remote SW/HW Reconfiguration of Nodes

In CaRDIN, the remote reconfiguration of an edge-centric node (at runtime) is automatically triggered upon changing the distributed classes (SW) or bitstream (HW) related to that node. This update process consist of installing new distriubted classes on the node's VM and flashing new bitstream (if needed) on the FPGA. The status of a node is identified based on the version number of related distributed classes

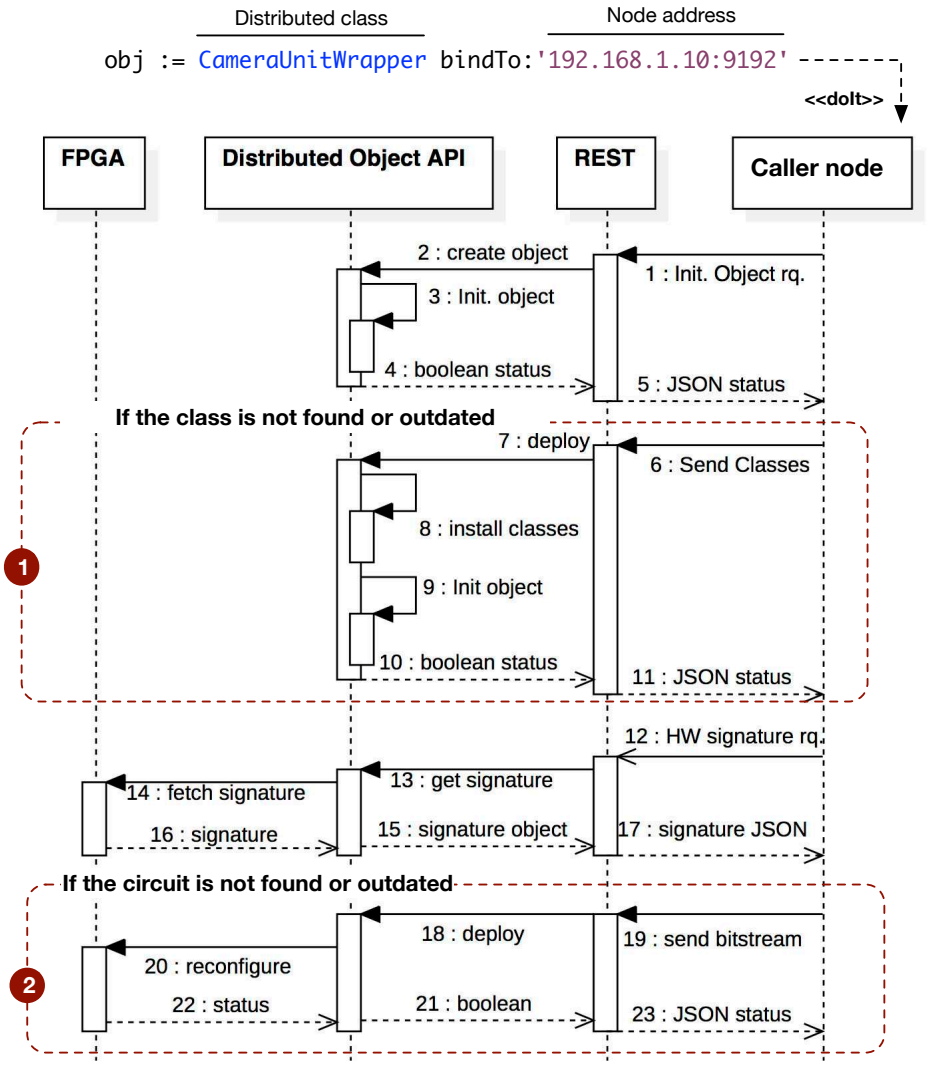


Figure 5.5: If the SW/HW is not deployed or outdated, the initialization of a distributed object will automatically trigger the reconfiguration of the node

and the signature of the current circuit on FPGA[‡]. The sequence diagram on figure 5.5 describes in detail this update process.

5.3.3 Discussion

The main ideas behind our middleware is *Centralization of code*, *automatic deployment* and *Collaboration execution*. *Centralization of code* facilitates the maintenance, management and development of distributed applications. All edge-centric nodes carry the same initial software setup with CaRDIN middleware pre-deployed. Application development is incremental and centralized on one place. New node's behaviour can be easily incorporated to the application by binding objects to the node. The DOA automatically handles the deployment and synchronisation of objects. A node can, at the same time, plays the role of a master(caller) or slave(callee) node. The middleware is dedicated to edge-centric computing and thus promotes the development of distributed algorithms on edge-centric nodes. *Collaboration execution* means the execution and the calculation resource of applications are diffused to the edge of the network, through transparently referencing and mixing surrogates for remote objects with local objects.

Our DOA shares some similarities with the base principle of the CORBA [OC12] specification, but indeed has some distinguished characteristics. Firstly, since we target a homogenous and centralized software environment on top of heterogenous hardware devices, a single language is used for distributed application development. Therefore there are no need for object interface definition (using an interface definition language and dedicated compiler). Client and server code coexist in a same distributed class so that the caller object and callee object naturally share the same interface. This simplifies the application development and reduces the middleware overhead on embedded system. Secondly, in our DOA, nodes on the network are equal (including the base station), a node can at the same time play the role of caller node and servant node (depending on the role of distributed objects). Therefore, nodes can coordinate between them without the present of a centralized server or name server. We rely on the REST protocol for object inter-communication. This allows us to benefit from the existing embedded HTTP server without adding additional software layer for the communication and thus results in a small footprint middleware.

While provide enough features to develop distributed application on the SN. This version of our DOA still has some limitations: (1) To successfully deploy a distributed class on a remote node, its supper class must be already existed on that node. Although the DOA allows to remotely deploy a specific class on the remote node, this must be manually handled by developer. (2) The error/exception mechanism is simple by returning an error object describing the stack trace of the remote VM. (3) There is not yet a mechanism to automatically manager the life cycle of remote objects. Our DOA allows to delete remote reference objects by manually performing remote delete (garbage) operation.

[‡]Assigned automatically by OoRCBridge toolset

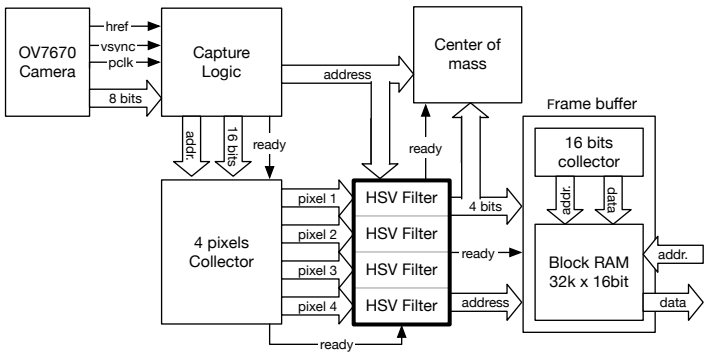


Figure 5.6: Object detection implementation on the FPGA.

5.4 Case Study 1: Camera Sensor Node Performing Image Processing

5.4.1 Scenario

This section describes an experiment that demonstrates and validates the proposed prototype platform. The CarDIN middleware is implemented to support the Smalltalk language but the principle applies to any dynamic language (Python, Ruby, etc.). The node's hardware is based on an Armadeus APF51 Single Board Computer which adopts a Freescale i.MX515 (Cortex-A8 @ 800MHz, 512MB DDR RAM) and a Xilinx Spartan 6 (LX9). The physical interface between the FPGA and the processor is WIEM (Wireless External Interface Module) with 16-bit dedicated data/address bus. The experiment is based on the image processing example presented in the previous chapter. It shows an implementation of a sensor node for image processing using our platform based on the FPGA circuit presented in the previous chapter. An OV7670 camera is connected directly to the GPIOs of the FPGA (APF51). The FPGA acquires image from camera and filters it using a HSV filter based on a color pattern. This filtered image is then used to estimate the position of object as the barycenter of the largest connected component. Figure 5.6 shows the simplified block diagram of the object detection circuit (at 100Mhz). This circuit can be configured (via dedicated flags) to work with either VGA or QVGA or QQVGA image. The base station and the node participate to the same LAN network using ethernet.

```
DeviceMapper subclass: #CameraUnitWrapper
CameraUnitWrapper >> gateway
  ^'ffvm/portal'
CameraUnitWrapper >> signature
  <remote>
  ^self int16At:18
CameraUnitWrapper >> x
  <remote>
  ^self int32At:24
CameraUnitWrapper >> y
  <remote>
  ^self int32At:28
"The following methods are manually added"
CameraUnitWrapper >> position
```

```
15    <remote>
16    ^{self x. self y}
17  CameraUnitWrapper >> positionDo:aCallbackBlock ns:anInt
18    <remote:#aCallbackBlock>
19    anInt timesRepeat:[
20      aCallbackBlock value: self position.
21      100 milliseconds wait.
22    ]
23  CameraUnitWrapper >> stream
24    self positionDo:[:p| p print] ns:500.
```

Listing 5.2: Example of a distributed class. The methods with a pragma are executed remotely. Others are locally executed methods

The image processing VHDL code is imported to the OORCBridge toolset for interface generation and virtual address mapping. The bitstream and the corresponding distributed class are then generated automatically and ready to be deployed on the node. Listing 5.2 shows the completed distributed class. The first 5 methods (lines 2-16) are generated automatically by the toolset. The last 2 methods (lines 17-24) are manually added for more complex features. At this point, no further development is needed for the node. In the developer point of view, the remote access to FPGA can be coded in a single class (11 active development lines). The system abstracts all the network communication, software partition, and hardware accessing that are performed transparently behind the code.

The API supports two kinds of pragma. The *<remote>* pragma on a method informs the caller that the method should be called remotely. When executing this kind of methods, the caller gathers all necessary arguments then passes them to the corresponding callee object. This latter will handle the execution and send back the result to the caller. The second kind of pragma, the *<remote:...>*, operates similarly to the first one, except that it allows to specify a callback for each returned data. This is especially helpful when continuously streaming data from the servant node to the caller node. The *#positionDo:ns:* (line 17) method in the listing 5.2 uses this kind of pragma. It allows to stream a number of samples of object position (specified by *anInt*) to the base station at a frequency of 10 Hz (100 milliseconds). Each sample will be handled by the callback *aCallbackBlock*, which is a Smalltalk block closure. Since the method is executed on callee object, the callback (line 20), on the opposite, is executed locally on caller object for every returned sample of data.

The *#gateway* method enables the caller object to identify the REST resource (url) of the sensor node. This is necessary when the caller intends to initialize a communication with the callee object. On the sensor node, all the annotated methods of *CameraUnitWrapper*, such as *#signature*,*#x*, *#y*, *#position*, and *#positionDo:ns:*, operate as local methods and have direct access to the FPGA register. On the base station, annotated methods (with *<remote>* or *<remote:...>*) are recognized as remote methods and will perform remote calls when executed. Others (without pragma) are normal Smalltalk methods and therefore will be performed locally on the base station.

The method *#stream* in line 23 is a normal Smalltalk method that uses a remote method in its body. The code requires the *#positionDo:ns:* method to stream 500 samples of object position from the node and then simply prints each one of these on the base station.

Table 5.1: Memory footprint(KB) of the web services

Module	Resident Set Size	Shared memory
Httpd	640	544
REST+VM	532	80

Resource	Object detection circuit
Slice registers	1,003/11,440 (8%)
Slice LUTs	1,551/5,720 (27%)
RAMB16BWERs	32/32 (100%)
BUFG/BUFGMUXs	3/16 (18%)

Table 5.2: FPGA resource used

5.4.2 Benchmarkings

At the beginning, when the node is powered on, only the software stack is running: the http server, the REST engine and the Smalltalk VM. Table 5.1 shows the static memory footprint of the software stack. The Resident Set Size points out the actual physical memory occupied by the software stack. This memory portion can be increased when the objects memory allocated for the Smalltalk image is full. In this case, the VM will claim for more dynamic memory allocation.

Table 5.3 shows the average percentage of the resources occupied by the software stack at different operation tests: (1) idle stage (with or without plugins). (2) The SW is deployed on the Smalltalk image. (3) The bitstream is configured on FPGA. (4) The base station continuously fetches 500 samples of object position from the node with the rate of 10Hz. (5) The node streams 500 samples of object data to the base station at a frequency of 10Hz.

At the idle stage (with or without plugins), the software stack has no impact on the CPU consumption. The HTTP server takes around 0,3% of memory, up to 0,5% when the system is fully loaded. The memory used is nearly constant when the node is in operation mode.

The hardware reconfiguration takes most of CPU resource in the 5 tests. This process consists in streaming the bitstream from the base station to the node, then deploying the bit stream on the FPGA. Since it invokes some kernel modules to communicate with the FPGA, the CPU has a work overhead (26,2%). After the reconfiguration, the FPGA starts its processing and is ready for communication. The table 5.2 shows the resource used by the object detection circuit on the FPGA.

Stage	Memory used(%)	CPU used(%)
(1) Httpd only (idle)	0.3	0
(1) Full system (idle)	0.5	0
(2) Software reconfig.	0.5	4.7
(3) Bitstream reconfig.	0.5	26.2
(4) Frequently fetching	0.55	21
(5) Streaming	0.5	5.3

Table 5.3: Resource used of the software stack in the node.

The last two tests carry the same operation: fetching 500 samples of object position from the node at the rate of 10 Hz. The *frequently fetching test* requires to open repeatedly a connection between the node and the base station (500 connections). These connections add more overhead to the CPU consumption (21%). The *streaming test* involves only one connection, then continuously streams data via that connection. Therefore, the process consumes averagely only 5% of the CPU resource.

Figure 5.7 shows the network load on the node in different active operations. Since the software/hardware reconfiguration requires files transfer, the bandwidth used for this operation goes up to 235 KiB/s for reception (RX) and 9.58 KiB/s for transmission (TX) in around 2 seconds. Others operations require the bandwidth of less than 20 KiB/s for RX and less than 5 KiB/s for TX. Obviously, the frequently fetching mode uses more network resource than the streaming mode. Above all, the fetching feature can be used to handle occasional requests since the network resource could be rapidly free at the end of a transmission. The streaming feature, in the other hand, is suitable for persistent and real-time connection, however, it requires that the network resource is reserved and blocked for a long time. There may be many connections on a node at the same time, therefore, the choice (fetching vs streaming) depends on the frequency of data that one wants to transmit.

This experiment uses an existing VHDL code for the image processing on the FPGA. But the scenario can be expanded by starting with a pure software image processing implementation (e.g. in C). Modern high level synthesis tools are able to synthesize this code to RTL. Our toolset can then handle the generated VHDL.

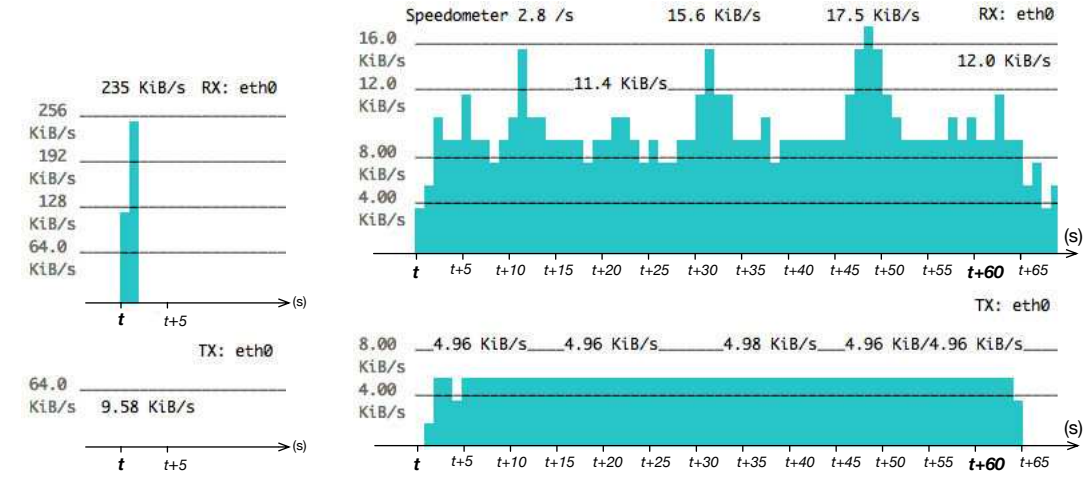
5.5 Case study 2: distributed algorithm development and deployment with CARDIN

To demonstrate the development and deployment of distributed algorithm on the network using CARDIN, we consider a simple camera surveillance scenario, as shown in figure 5.8: 3 camera sensor (hybrid) nodes are deployed on the network, they cary the same operation that tracks a moving ball based on a colour pattern. At anytime, the base station can ask any one of them about the node that actually has the ball. These nodes need to coordinate between them to decide who is handling the ball. This decision could be based on the size of the ball seen by each camera. The node with the largest detected zone will be voted for the leader.

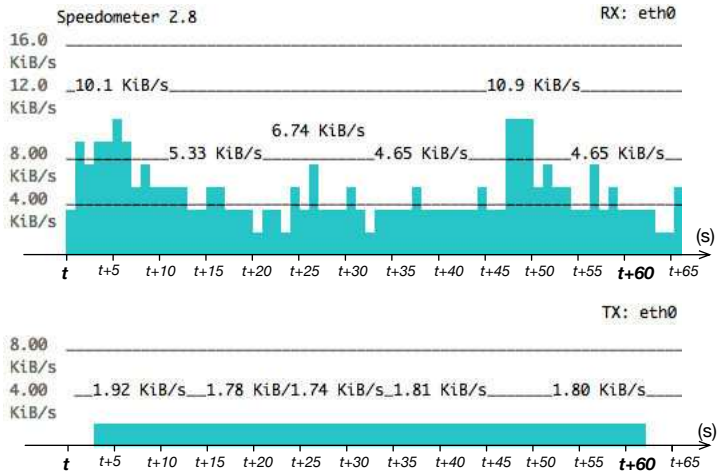
The three camera sensor nodes are based on the same setup of the previous case study (section 5.4). For the detection circuit, to calculate the size of the detected zone, we replace the frame buffer unit with a pixel counter unit, as shown in figure 5.9. This latter will count the total number of the filtered pixels as the size of the detected zone. This value can be query from software via CARDIN middleware.

For the voting algorithm, we base on a simple Token Ring Election algorithm[§] as shown in algorithm 1 [Ray13]. Concretely, each node on the network has a unique ID and a reference to the next node. The last node refers to the first node to constitute a ring. The same token ring election algorithm is implemented on each node. When the base station wants to know the leader (e.g. the node which has the ball), it demands a node to start the election, an election message is created and sent from node to node until it reaches the original sender (the starter node). At this point, the decision is made to vote

[§] http://www.cs.colostate.edu/ cs551/CourseNotes/Synchronization/RingElectExample.html



(a) (b)



(c)

Figure 5.7: Network load of the node on different operations: (5.7a) the software/bitstream reconfiguration process ($[t, t+2]$); (5.7b) the frequently fetching test ($[t, t+70]$) and lastly (5.7c) the streaming test ($[t, t+70]$). t is the time when an operation begins.

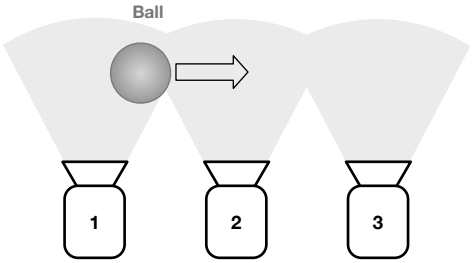


Figure 5.8: 3 camera sensor nodes tracking a moving ball. Question: Which camera actually has the ball?

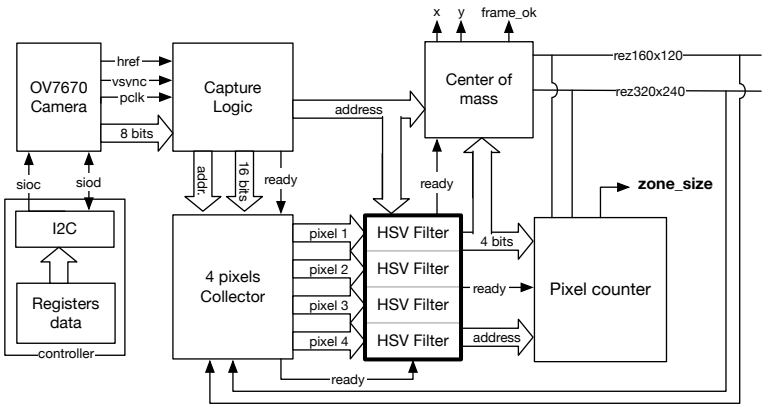


Figure 5.9: The frame buffer unit is replaced by the pixel counter unit to count the filtered pixels.

for the leader. A coordinator message will then be returned to each node to acknowledge the new leader. This election process is happened between sensor nodes without passing to the base station.

Data: $passedNodes = \{N_i = \{ID, token\} | i = 0..n\}$
Result: The elected node $N = \{ID, token\}$
$N_c = \{ID, token\}$ is the identity of the current node;
if *passedNodes includes* N_c **then**
    leader := N_i with $N_i.token = max(\{N_j.token | j = 1..n\})$;
    return leader;
else
    N_c := query the identity of the current node;
    Add N_c to *passedNodes*;
    leader := send *passedNodes* to the next node for electing;
    Acknowledge *leader*;
    return *leader*;
end

Algorithm 1: Simple Token Ring Election Algorithm

To implement this algorithm in CaRDIN, we first import the image processing VHDL code to the OORCBridge toolset to generate the bitstream and the corresponding distributed class. This class is then used to implement the election mechanism (by adding additional methods to it). The entire distributed application can be coded in a single class as shown in listing 5.3. The Token Ring Election Algorithm is implemented in the method *#elect:* which is defined as a remote method.

```
1  DeviceMapper subclass: #CameraUnitWrapper
2  ...
3  CameraUnitWrapper >> zone_size
4    <remote>
5    ^self int32At:32
6  CameraUnitWrapper >> id:v
7    id := v
8  CameraUnitWrapper >> nextNode:v
9    nextNode := v
10 CameraUnitWrapper >> elect:electors
11   <remote>
12   |arr leader role|
13     arr := electors collect:[:e| e at:1].
14     (arr includes: id) ifTrue:[
15     "The message has gone around the ring. Time for the leader decision"
16         leader := electors at:1.
17         electors do:[ :e | ((e at:2) > (leader at:2)) ifTrue:[leader := e]].
18         ^leader
19     ] ifFalse:[
20       "Backup the role of the next node"
21         role := nextNode master.
22         "Set the next node as master, so that it can make a remote call"
23         nextNode master:true.
24         "Passing the message to the next node"
25         leader := nextNode elect:(electors with:{id. self zone_size}).
26         "restore the role of the next node"
27         nextNode master:role.
28         ^leader
29     ]
```

Listing 5.3: Token Ring Implementation for camera surveillance examples using CaRDIN

Figure 5.10 shows how the application is deployed and run on the network. On the base station, three objects of the class *CameraUnitWrapper* are created and bound to the 3 camera sensor nodes. The ring is built by associating a successor node to each node. The base station then demands the *node1* to start the election by performing a remote call via the method *#elect:*. At the first run, the distributed class and the associated bitstream are automatically deployed on the corresponding node before each execution. This process is repeated when a node require its successor to perform the *#elect:* method (line 25 of the listing 5.3). This causes an automatic deployment chain of the application on all the three nodes on the network.

In our DOA, all nodes are equal, a node can play at the same time the role of a caller node and a callee node. That is, a node can handle a remote call while being able to perform remote calls to other nodes. For example, while the node 1 executes the *#elect:* method required by the base station, it can passing the election message to the node 2 by performing a remote call of *#elect:* to that node, and so on. Therefore, sensor nodes are able to communicate with each other without passing to the base station. Note that at line 23 of the listing 5.3, we need to set the role of the current's successor object

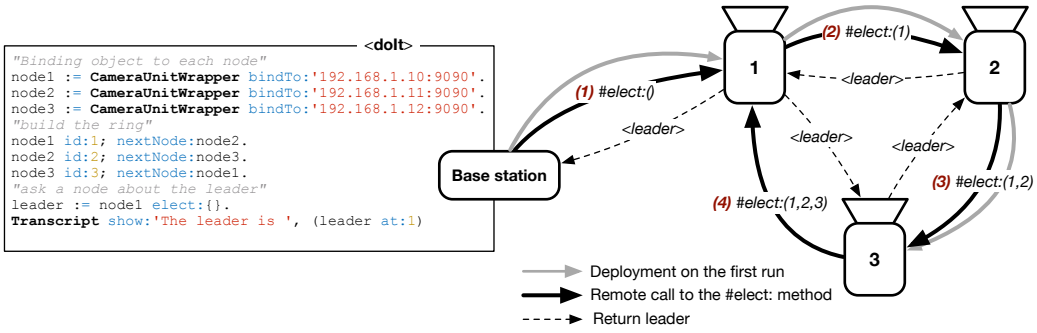


Figure 5.10: Deployment and execution of the distributed application via CaRDIN middleware.

to a master object, so that it can actually perform an remote call to the next node.

This case study show how complex distributed algorithms can be developed and deployed using CaRDIN. It demonstrates the main philosophy behind our DOA: *centralization development*, *automatic deployment* and *distributed execution*. Firstly, the entire application are written on the base station. Our DOA allows developers to focus on the algorithm implementation without worrying about network communication, code deployment and node-to-node coordination. Secondly, the deployment of the application on the network is at runtime and automatic. This process can happen between nodes without the intervention of the base station. Finally, the middleware encourages the independent coordination between nodes, which is important in edge-centric applications where nodes may coordinate between them to make a decision instead of relying on centralized server. Futhermore, new node can be easily incorporated into the system by simply binding it to remote objects.

5.6 Summary

In this chapter, we introduced CaRDIN, a platform for efficiently developing and deploying IP-based edge-centric network. Our proposal uses an FPGA, that offers high performance while reducing power consumption. We show that an hybrid hardware system (FPGA + processor) along with a web service oriented software platform enables remote reconfiguration/reprogramming of sensor nodes. The use of web services favors straightforward integration with existing IoT standards. Moreover, we rely on an object-oriented language to support transparently referencing remote objects. The Virtual machine brings the dynamicity of interpreted languages and allow the development of distributed algorithms on the node, through mixing surrogates for remote objects with local objects. The distributed software API hides all the details of node operations and interactions and let developers focus on the functionalities of their application.

To demonstrate the proposed platform, in the fist case study, we have built a smart sensor node on an APF51 board as a proof of concept. This node connects to a camera and performs an object detection algorithm on the FPGA. It communicates the object position to the base station using the distributed objects API. The second case study demonstrates the use of CarDIN to develop and development distributed applications on the network which allow node-to-node coordination.

La conclusion résulte souvent de ce moment précis où vous en avez eu marre...

–Anonyme

6 Conclusion and Perspectives

Contents

6.1 Contribution Summary

The research in this thesis studies the application of modern design methodology, in particular OOD, on embedded system. The context here converges to the utilization of FPGA/processor hybrid devices in IoT and edge-computing. This results in a propositional of a dedicated design flow, middleware and toolsets for simplifying the SW/HW development process. The ultimate goal is to provide an environment that allows end users to benefit from OOD to easily develop edge-centric applications, while always have the ability to exploit the hardware features of FPGAs in a software-friendly way. Therefore, our contribution address the targeted problem following three main axes:

Promoting OOD on HW design with the OoRC meta-model

The OoRC meta-model is designed with OOD principles and design pattern in mind. While it fully support modelling FPGA circuits at RTL (fine-grained) level (using the dedicated DSL), its main objective is to enhance the HW system-level (coarse-grained) design experience. The philosophy here is to use abstraction to separate as much as possible the problem space from the implementation space. Thus, one can use the meta-model to provide a generic solution (i.e. implementation-independent) to a specific problem so that it can be adaptable in different application contexts. This encourages the intensive use of modularity, adaptability and extensibility concepts in hardware design to increase system reusability and hence improve productivity. One advantage of our meta-model compared to existing approaches is that it supports reuse of VHDL legacy IPs as regular object oriented models. This

feature extends the ability of the meta-model to manipulate existing third party IPs, such as performing automatic processing, integration, structural refactoring or feature injection, etc.

The meta-model can be employed as the base model for different design purposes. For example, it can be used to develop CAD tools, to build higher level graphical modelling tool (e.g. UML) or to design a HW/SW co-design environment/platform, etc.

Bridging the HW/SW interfacing gap with OoRCBridge

OoRCBridge is a platform built on top of the OoRC meta-model. It uses OOD combined with the platform-based methodology and the early-binding approach for SW/HW co-design. It aims at integrating FPGA devices with existing high level object oriented software. The key point of OoRCBridge is *independent integration* via interface standadization. Both SW and HW (FPGA) parts of the system comply with a standadized communication interface and protocol. They can therefore freely vary with respect to this agreement, regardless of the underlying physical architecture. The standadized interface is architecture specific. Fortunately, its implementation can be abstracted and automatized using the *correct-by-construction* and *generation* techniques. An interface model (template) can be constructed in an abstract form. It is used to reason about the proposed implementation of the interface depending on the inputs, then ensuring that all required functionality will be delivered and the correct behaviour exhibited. Automatic code generation closes the SW/HW gap and provides software-friendly API for hardware accessing from high-level software environment.

Centralized development and automatic remote deployment of edge-centric applications with CaRDIN

CaRDIN provides a dedicated environment for developing edge-centric applications on the network (IP-based) of hybrid FPGA/processor devices. It uses OoRCBridge to facilitates and standardizes the HW/SW integration on each node. On top of this, a distributed software environment is set up using a VM approach. The proposed DOA allows to centralize the development process by mean of transparently referencing and mixing surrogates for remote objects with local objects. On the one hand, this feature favours the application development, management and maintenance. On the other hand, it simplifies the incorporation of new nodes into the application using the automatic object binding mechanism. In addition, SW/HW on the nodes can be remotely reconfigured without the need of manually detaching them from the deployment site. This supports better system scalability seeing that edge-centric applications may rely on a large-scale network.

To demonstrate and validate the proposed prototype platform (CaRDIN), an experiment based on sensor node that perform image process has been built, as a proof of concept. This node connects to a camera and performs an object detection algorithm on the FPGA. It communicates the object position to the base station using our distributed objects API.

6.2 Current and Future Works

Regarding the OoRC meta-model, beside applying OOD on hardware design, reuse of existing third party VHDL legacy is also an important feature, since it can reduce the design cost and time. For the moment, we have only checked the validity of the VHDL parser, which transforms the VHDL code to an abstract syntax tree (AST). However, the transformation of ASTs to OoRC circuit models still remains unverified*. Therefore, the next step is finding methods and benchmarks to verify the correctness this transformation. This involves additional work on the model-checking domain for a formal verification.

The meta-model can be further abstracted by adding more abstraction layers on top of it. Apart from OoRCBridge, we plan to study other use-cases in hardware design. It can be used to build high-level CAD tool (GUI), that can automatize the design process at certain level. Or, it can be combined with other system-level design methodology, such as UML, to provide high-level graphical hardware specification and SW/HW co-design environment.

As for the CaRDIN environment, while it is possible to dynamically reconfigure software on a node, the remote hardware reconfiguration (FPGA) is often done in an offline manner. That is, the entire FPGA needs to be reconfigured and reset even if the change only affects a small part of the circuit. This can cause an interruption on other independent parts that are processing data. A solution to this problem is adding FPGA partial reconfiguration support to the node's middleware. This feature allows to make change only on a part of the *FPGA* while conserving the others part from resetting.

The current version of our DOA still has some limitations as presented in section 5.3.3. These limitations could be the future features of the DOA. Concretely: (1) we can add classes dependencies mechanism to the DOA, when a distributed class is deployed on a node, the system automatically deploys also all its missing dependency classes.(2) The DOA supports automatic garbage collection of remote objects. This can be done at VM level. The idea is when the VM reclaims the memory occupied by a caller object (that is no longer in use by the program), it also perform a remote call to the corresponding remote VM to delete the referenced object of the servant node.

The edge-centric experiments in this thesis is only a proof of concept. The ultimate goal is to use the prototype platform to deploy truly distributed applications on an edge-centric network (e.g. surveillance). Such sensor network consists of many smart nodes and has the capacity to simply incorporate new nodes into the system. Then different strategies can be performed and tested to optimize the distribution of calculation resource on the network. Concretely, the distribution of an application on edge-centric network should be scheduled based on the resource –such as memory, CPU, or battery status– available on each node. This may lead to the development of a task scheduler on the CaRDIN middleware that allows to efficient delivery calculation tasks to the edge of the network, while away balancing the energy/power constraint on each node. In software domain, this relates to the work on dynamic software update which targets solutions for dynamically migrating code between nodes [TPF+16]. Moreover, it becomes more interesting if one can not only migrate code between nodes, but also the execution state of the nodes. With the VM, it is trivial to take a snapshot of a software on a node then execute it on other nodes. As for the FPGA, we can rely on a hardware virtualization

*In fact, the validation can be manually performed by re-exporting the imported legacy model to VHDL and then comparing the two VHDL codes

solution. There are actually some remarkable works on this topic, such as [LLLB14, BNLLL16] that allow to capture the current execution state of an FPGA then continue it on other FPGAs using a virtual FPGA architecture. Security and Privacy of sensor data on the network are another interesting research topic. In traditional SN, outputs of sensor nodes are often vulnerable to unauthorized observation. When these nodes participate to the internet (IoT), this become a serious problem since the sensor data can be easily sniffed by enemy/third-party application over the internet. This raises the need of a cryptography mechanism on SN [GKS05, AB08]. Cryptography is complex, slow and power hungry, thus is not alway suitable to use on traditional sensor nodes. However, our proposed edge-centric nodes can overcome this constraint with the help of FPGA. Cryptography algorithm can be implemented on real hardware (FPGA) which allows fast execution and power friendly.

Bibliography

[AB08] Tuncer Can Aysal and Kenneth E Barner. Sensor data cryptography in wireless sensor networks. *IEEE Transactions on Information Forensics and Security*, 3(2):273–289, 2008.

[Alb05] C. Albrecht. Iwls 2005 benchmarks. *2005 International Workshop on Logic Synthesis*, June 2005.

[Arm] Armadeus. APF51 Single Board Computer.

[BAS14] G. Bazydlo, M. Adamski, and Ł. Stefanowicz. Translation uml diagrams into verilog. In *2014 7th International Conference on Human System Interactions (HSI)*, pages 267–271, June 2014.

[BdDPP16] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescapé. Integration of cloud computing and internet of things: a survey. *Future Generation Computer Systems*, 56:684–700, 2016.

[Ber12] Gonçalo Bernardo. *Online deployment of dependent tasks onto networked systems*. PhD thesis, TU Delft, Delft University of Technology, 2012.

[Béz05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[BH98] P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, 1998.

[BHSS07] Athanassios Boulis, Chih-Chieh Han, Roy Shea, and Mani B Srivastava. Sensorware: Programming sensor networks beyond code update and querying. *Pervasive and Mobile Computing*, 3(4):386–412, 2007.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[BNLLL16] Théotime Bollengier, Mohamad Najem, Jean-Christophe Le Lann, and Loïc Lagadec. Zeff: Une plateforme pour l'intégration d'architectures overlay dans le cloud. In *COMPAS 2016*, 2016.

[BRS13] David F. Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Commun. ACM*, 56(4):56–63, April 2013.

[BSP05] Andrew Bainbridge-Smith and Su-Hyun Park. Adh: an aspect described hardware programming language. In *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.*, pages 283–284. IEEE, 2005.

[CCA+13] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24, 2013.

[CCD+09] Eduardo Canete, Jaime Chen, Manuel Diaz, Luis Llopis, and Bartolome Rubio. A service-oriented middleware for wireless sensor and actor networks. In *Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on*, pages 575–580. IEEE, 2009.

[CF14] José Cecílio and Pedro Furtado. *Wireless Sensors in Heterogeneous Networked Systems*. 2014.

[CF16] Alessandro Cilardo and Edoardo Fusella. Design automation for application-specific on-chip interconnects: A survey. *Integration, the VLSI Journal*, 52:102–121, 2016.

[CRS00] F. Corno, M.S. Reorda, and G. Squillero. Rt-level itc'99 benchmarks and first atpg results. *Design Test of Computers, IEEE*, 17(3):44–53, Jul 2000.

[CS03] Cincom-Systems. *VisualWorks ® Distributed Smalltalk Application Developer ' s Guide*. 2003.

[CSM08] G. Chalivendra, R. Srinivasan, and N. S. Murthy. Fpga based re-configurable wireless sensor network protocol. In *2008 International Conference on Electronic Design*, pages 1–4, Dec 2008.

[CT05] Frank P Coyle and Mitchell A Thornton. From uml to hdl: a model driven architectural approach to hardware-software co-design. 2005.

[CWFH13] Christopher Cullinan, Christopher Wyant, Timothy Frattesi, and Xinming Huang. Computing performance benchmarks among cpu, gpu, and fpga. *Internet: www. wpi. edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking Final*, 2013.

[D+09] Adam Dunkels et al. Efficient application integration in ip-based sensor networks. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 43–48. ACM, 2009.

[DA13] M. Doligalski and M. Adamski. Uml state machine implementation in fpga devices by means of dual model and verilog. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 177–184, July 2013.

[Dam06] R Damaševičius. On the application of meta-design techniques in hardware design domain. *International Journal of Computer Science (IJCS)*, 1(1):67–77, 2006.

[Dec04] Jan Decaluwe. Myhdl: a python-based hardware description language. *Linux journal*, 2004(127):5, 2004.

[DLPBT12] Antonio De La Piedra, An Braeken, and Abdellah Touhafi. Sensor systems based on fpgas and their applications: A survey. *Sensors*, 12(9):12235–12264, 2012.

[DMŠ03] Robertas Damaševičius, Giedrius Majauskas, and Vytautas Štuikys. Application of design patterns for hardware design. In *Proceedings of the 40th annual Design Automation Conference*, pages 48–53. ACM, 2003.

[DS04] Robertas Damasevicius and Vytautas Stuikys. Application of uml for hardware design based on design process model. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 244–249, Piscataway, NJ, USA, 2004. IEEE Press.

[DZHC17] Stéphane Ducasse, Dmitri Zagidulin, Nicolai Hess, and Dimitris Chloupis. *Pharo by Example 50*. Lulu.com & Square Bracket Associates, 2017.

[EBZ+12] Majdi Elhaji, Pierre Boulet, Abdelkrim Zitouni, Samy Meftali, Jean-Luc Dekeyser, and Rached Tourki. System level modeling methodology of noc design from uml-marte to vhdl. *Des. Autom. Embedded Syst.*, 16(4):161–187, November 2012.

[EFQ15] Emad Ebeid, Franco Fummi, and Davide Quaglia. Hdl code generation from uml/marte sequence diagrams for verification and synthesis. *Design automation for embedded systems*, 19(3):277–299, 2015.

[ELLSV01] Stephen Edwards, Luciano Lavagno, Edward A Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Readings in hardware/software co-design*, 86, 2001.

[Eva11] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1(2011):1–11, 2011.

[Fab07] Luc Fabresse. *From Decoupling to Unanticipated Component Assembly: Design and Implementation of the Component-Oriented Language Scl*. Theses, Université Montpellier II - Sciences et Techniques du Languedoc, December 2007.

[FB01] Robert France and James Bieman. Multi-view software evolution: a uml-based framework for evolving object-oriented software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 386. IEEE Computer Society, 2001.

[FCC+14] Blair Fort, Andrew Canis, Jongsok Choi, Nazanin Calagar, Ruolong Lian, Stefan Hadjis, Yu Ting Chen, Mathew Hall, Bain Syrowik, Tomasz Czajkowski, et al. Automating the design of processor/accelerator embedded systems with legup high-level synthesis. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 120–129. IEEE, 2014.

[FFH+11a] Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. A service component framework for multi-user scenario management in ubiquitous environments. In *ICSEA'2011: 6th International Conference on Software Engineering Advances*, page N/A, Barcelona, Spain, October 2011.

[FFH+11b] Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. User-defined scenarios in ubiquitous environments: creation, execution control and sharing.

In *SEKE: Software Engineering and Knowledge Engineering*, pages 302–307, Miami, United States, July 2011.

[Fri13] Peter Friess. *Internet of things: converging technologies for smart environments and integrated ecosystems*. River Publishers, 2013.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GKS05] Gunnar Gaubatz, Jens-Peter Kaps, and Berk Sunar. *Public Key Cryptography in Sensor Networks—Revisited*, pages 2–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[GL95] Shaori Guo and Wayne Luk. Compiling ruby into fpgas. In *International Workshop on Field Programmable Logic and Applications*, pages 188–197. Springer, 1995.

[GLBP+11] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embed. Comput. Syst.*, 10(4):39:1–39:36, November 2011.

[GLME+15] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[GRL+08] Levent Gurgen, Claudia Roncancio, Cyril Labbé, André Bottaro, and Vincent Olive. Sstreamware: a service oriented middleware for heterogeneous sensor data management. In *Proceedings of the 5th international conference on Pervasive services*, pages 121–130. ACM, 2008.

[HBH+99] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A CAD suite for high-performance FPGA design. *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00375)*, 1999.

[HKT+07] Wolfgang Haberl, Stefan Kugele, Michael Tautschnig, Andreas Bauer, Christian Schallhart, Stefano Merenda, Christian Kühnel, Florian Müller, Zhonglei Wang, Doris Wild, et al. Cola–the component language. 2007.

[HRVG08] H. Hinkelmann, A. Reinhardt, S. Varyani, and M. Glesner. A reconfigurable prototyping platform for smart sensor networks. In *2008 4th Southern Conference on Programmable Logic*, pages 125–130, March 2008.

[IEE04] IEEE. IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. Technical Report October, 2004.

[IEE14] IEEE. IEEE Standard for IP-Xact, Standard structure for packaging, integrating, and reusing IP within Tool Flows. Technical report, IEEE Computer Society, 2014.

[int11] International technology roadmap for semiconductors, 2011.

[JDM+07] A. Jerraya, J.M. Daveau, G. Marchioro, C. Valderrama, M. Romdhani, T. Ben Ismail, N.E. Zergainoh, F. Hessel, P. Coste, Ph. Le Marrec, A. Baghdadi, and L. Gauthier. *Hardware/Software co-design*, pages 133–158. Springer US, Boston, MA, 2007.

[Jéz08] Jean-Marc Jézéquel. Model driven design and aspect weaving. *Software & Systems Modeling*, 7(2):209–218, 2008.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, pages 128–138, Berlin, Heidelberg, 2006. Springer-Verlag.

[KBLK07] T. Kobialka, R. Buyya, C. Leckie, and R. Kotagiri. A sensor web middleware with stateful services for heterogeneous sensor networks. In *2007 3rd International Conference on Intelligent Sensors, Sensor Networks and Information*, pages 491–496, Dec 2007.

[Kla07] Benjamin Klatt. Xpand: A closer look at the model2text transformation language. *Language*, 10(16):2008, 2007.

[KNRSV00] Kurt Keutzer, A Richard Newton, Jan M Rabaey, and Alberto Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

[KPC+08] YE Krasteva, J Portilla, JM Carnicer, E de la Torre, and T Riesgo. Remote hw-sw reconfigurable wireless sensor nodes. In *Industrial Electronics, 2008. IECON 2008. 34th Annual Conference of IEEE*, pages 2483–2488. IEEE, 2008.

[KS09] Kavi Kumar Khedo and RK Subramanian. A service-oriented component-based middleware architecture for wireless sensor networks. 2009.

[KTO16] L. Kechiche, L. Touil, and B. Ouni. Real-time image and video processing: Method and architecture. In *2016 2nd International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*, pages 194–199, March 2016.

[Küh05] Thomas Kühne. What is a model? In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2005.

[LBMD08] Sébastien Le Beux, Philippe Marquet, and Jean-Luc Dekeyser. Model Driven Engineering Benefits for High Level Synthesis. Research Report RR-6615, INRIA, 2008.

[Lee08] E. A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, May 2008.

[LFL+16a] Xuan Sang LE, Luc Fabresse, Jannik Laval, Jean-Christophe Le Lann, Loïc Lagadec, and Noury Bouraqadi. Dynamic distributed programming on reconfigurable ip-based smart sensor networks. Presented as poster at 11ème Colloque du GDR SoC-SiP, France, 2016.

[LFL+16b] Xuan Sang LE, Luc Fabresse, Jannik Laval, Jean-Christophe Le Lann, Loïc Lagadec, and Noury Bouraqadi. Speeding Up Robot Control Software Through Seamless Integration With FPGA. In *SHARC '16: 11th National Conference on Software and Hardware Architectures for Robots Control*, Brest, France, 2016.

[LHKS91] John A. Lewis, Sallie M. Henry, Dennis G. Kafura, and Robert S. Schulman. An empirical study of the object-oriented paradigm and software reuse. *SIGPLAN Not.*, 26(11):184–196, November 1991.

[LLF+14] Xuan Sang LE, Loïc Lagadec, Luc Fabresse, Jannik Laval, and Noury Bouraqadi. From Smalltalk to Silicon: Towards a methodology to turn Smalltalk code into FPGA. In *IWST 14*, Cambridge, United Kingdom, August 2014.

[LLF+15] Xuan Sang LE, Loïc Lagadec, Luc Fabresse, Jannik Laval, and Noury Bouraqadi. A meta model supporting both hardware and smalltalk-based execution of fpga circuits. *IWST '15*, pages 6:1–6:14, 2015.

[LLLB14] Loic Lagadec, Jean-Christophe Le Lann, and Theotime Bollengier. A prototyping platform for virtual reconfigurable units. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, pages 1–7. IEEE, 2014.

[LLLL+16] Xuan Sang LE, Jean-Christophe Le Lann, Loïc Lagadec, Luc Fabresse, Noury Bouraqadi, and Jannik Laval. Cardin: An agile environment for edge computing on reconfigurable sensor networks. In *the proceedings of The 2016 International Conference on Computational Science and Computational Intelligence (CSCI'16)*, Las Vegas, Nevada, USA, 2016.

[LMP+05] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[LSKT13] Junsong Liao, Brajendra K Singh, Mohammed AS Khalid, and Kemal E Tepe. Fpga based wireless sensor node with customizable event-driven architecture. *EURASIP Journal on Embedded Systems*, 2013(1):1, 2013.

[LSS13a] Anders B Lange, Ulrik P Schultz, and Anders S Soerensen. Unity: A unified software/hardware framework for rapid prototyping of experimental robot controllers using fpgas. In *Proc. of the 8th full-day Workshop on Software Development and Integration in Robotics, Karlsruhe, Germany*, 2013.

[LSS13b] Andre B Lange, Ulrik Pagh Schultz, and Anders Stengaard Soerensen. Unity-link: A software-gateware interface for rapid prototyping of experimental robot controllers on fpgas. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 3899–3906. IEEE, 2013.

[LSS14] Anders Blaabjerg Lange, Ulrik Pagh Schultz, and Anders Stengaard Soerensen. Towards Automatic Migration of ROS Components from Software to Hardware. 2014.

[LW16a] Marcela Leite and Marco Aurélio Wehrmeister. System-level design based on uml/-marte for fpga-based embedded real-time systems. *Design Automation for Embedded Systems*, 20(2):127–153, 2016.

[LW16b] Marcela Leite and Marco Aurélio Wehrmeister. System-level design based on uml/-marte for fpga-based embedded real-time systems. *Design Automation for Embedded Systems*, 20(2):127–153, 2016.

[MAK07] Rene Mueller, Gustavo Alonso, and Donald Kossmann. Swissqm: Next generation data processing in sensor networks. 2007.

[Mar16] Umit Y Ogras Radu Marculescu. Communication-based design for nanoscale socs. *The VLSI Handbook*, 2016.

[MC99] W. E. McUmber and B. H. C. Cheng. Uml-based analysis of embedded systems using a mapping to vhdl. In *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on*, pages 56–63, 1999.

[MFHH05] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[MG11] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[MGSPF11] T. R. Muck, M. Gernoth, W. Schroder-Preikschat, and A. A. Frohlich. A case study of aop and oop applied to digital hardware design. In *2011 Brazilian Symposium on Computing System Engineering*, pages 66–71, Nov 2011.

[MPV11] L. Mainetti, L. Patrono, and A. Vilei. Evolution of wireless sensor networks towards the internet of things: A survey. In *SoftCOM 2011, 19th International Conference on Software, Telecommunications and Computer Networks*, pages 1–6, Sept 2011.

[MR08] P Muralidhar and CB Rama Rao. Reconfigurable wireless sensor network node based on nios core. In *Wireless Communication and Sensor Networks, 2008. WCSN 2008. Fourth International Conference on*, pages 67–72. IEEE, 2008.

[MS09] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, 26(4):18–25, July 2009.

[MVG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[MVG+12] Wim Meeus, Kristof Van Beeck, Toon Goedeme, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, August 2012.

[MWP+10] Tomas G Moreira, Marco A Wehrmeister, Carlos E Pereira, Jean-Francois Petin, and Eric Levrat. Automatic code generation for embedded systems: From uml specifications to vhdl code. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 1085–1090. IEEE, 2010.

[Nga11] Nicolas Ngan. *Etude et conception d'un réseau sur puce dynamiquement adaptable pour la vision embarquée*. PhD thesis, Paris Est, 2011.

[OC12] OMG-CORBA. Common Object Request Broker Architecture (CORBA) Specification, Version 3.3. Technical report, Object Management Group, 2012.

[OMG14] OMG. Moden Driven Architecture (MDA) guide rev. 2.0. Technical report, Object Management Group, 2014.

[OMG15] OMG. Meta Object Facility (MOF) Core Specification v2.5. Technical report, Object Management Group, 2015.

[OMG16] OMG. Meta Objet Facility (MOF) 2.0 Query/View/Transformation, V1.3. Technical report, Object Management Group, 2016.

[Ora12] Oracle. Java RMI Release Notes, 2012.

[ORRM09] Frank Oldewurtel, Janne Riihijarvi, Krisakorn Rerkrai, and Petri Mahonen. The runes architecture for reconfigurable embedded and sensor networks. In *Sensor Technologies and Applications, 2009. SENSORCOMM'09. Third International Conference on*, pages 109–116. IEEE, 2009.

[OU11] OMG-UML/MARTE. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Technical report, Object Management Group, 2011.

[OU15] OMG-UML. OMG Unified Modeling Language TM (OMG UML). Technical report, Object Management Group, 2015.

[OWTK10] A. Oetken, S. Wildermann, J. Teich, and D. Koch. A bus-based soc architecture for flexible module placement on reconfigurable fpgas. In *2010 International Conference on Field Programmable Logic and Applications*, pages 234–239, Aug 2010.

[PBMB16] Maxime Pelcat, Cédric Bourrasset, Luca Maggiani, and François Berry. Design Productivity of a High Level Synthesis Compiler versus HDL. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2016)*, Agios Konstantinos, SAMOS, Greece, July 2016.

[PBSV+04] Alessandro Pinto, Alvise Bonivento, Allberto L Sangiovanni-Vincentelli, Roberto Passerone, and Marco Sgroi. System level design paradigms: Platform-based design and communication synthesis. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, volume 11, pages 537–563. ACM, 2004.

[PBV07] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. The molen compiler for reconfigurable processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(1):6, 2007.

[PF11] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the ANTLR parser generator. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 425–436, 2011.

[PKGZ08] Nissanka B Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 253–266. ACM, 2008.

[PL08] Damien Picard and Loic Lagadec. Multi-Level Simulation of Heterogeneous Reconfigurable Platforms. *ReCoSoC'08, Barcelona, Spain*, 2008.

[Pom16] Luigi Pomante. *Electronic System-Level HW/SW Co-Design of Heterogeneous Multi-Processor Embedded Systems*. River Publishers, 2016.

[PRDC] Jorge Portilla, Teresa Riesgo, and Angel De Castro. A reconfigurable fpga-based architecture for modular nodes in wireless sensor networks. In *2007 3rd Southern Conference on Programmable Logic*, pages 203–206. IEEE.

[QMD08] I. R. Quadri, S. Meftali, and J. L. Dekeyser. Marte based modeling approach for partial dynamic reconfigurable fpgas. In *2008 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, pages 47–52, Oct 2008.

[RA12] Jean-Claude Royer and Hugh Arboleda. *Model-Driven and Software Product Line Engineering (ISTE)*. Wiley-IEEE Press, 1st edition, 2012.

[Ray13] Michel Raynal. *Leader Election Algorithms*, pages 77–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Ren14] Haoxing Ren. A brief introduction on contemporary high-level synthesis. In *2014 IEEE International Conference on IC Design & Technology*, 2014.

[RHTO15] Ciprian-Radu Rad, Olimpiu Hancu, Ioana-Alexandra Takacs, and Gheorghe Olteanu. Smart monitoring of potato crop: a cyber-physical system architecture model in the field of precision agriculture. *Agriculture and Agricultural Science Procedia*, 6:73–79, 2015.

[RSRB05] Elvinia Riccobene, Patrizia Scandurra, Alberto Rosti, and Sara Bocchio. A soc design methodology involving a uml 2.0 profile for systemc. In *Design, Automation and test in Europe*, pages 704–709. IEEE, 2005.

[SC95] Douglas C. Schmidt and Charles D. Cranor. Half-sync/half-async - an architectural pattern for efficient and well-structured concurrent i/o. *in Proceedings of the 2 nd Annual Conference on the Pattern Languages of Programs*, pages 1–10, 1995.

[SCC+06] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java™on the bare metal of wireless sensor devices: The squawk java virtual machine. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM.

[ŠD13] Vytautas Štuikys and Robertas Damaševičius. *Applications of Meta-Programming Methodology*, pages 291–316. Springer London, London, 2013.

[SDTF12] Petr Spacek, Christophe Dony, Chouki Tibermacine, and Luc Fabresse. An inheritance system for structural and behavioral reuse in component-based software programming. *SIGPLAN Not.*, 48(3):60–69, September 2012.

[SDTF13] Petr Spacek, Christophe Dony, Chouki Tibermacine, and Luc Fabresse. Wringing out objects for programming and modeling component-based systems. In *Proceedings of the Second International Workshop on Combined Object-Oriented Modelling and Programming Languages*, ECOOP'13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[Sed06] Nicholas Peter Sedcole. *Reconfigurable platform-based design in FPGAs for video image processing*. 2006.

[SF16] Sima Sinaei and Omid Fatemi. Novel heuristic mapping algorithms for design space exploration of multiprocessor embedded architectures. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 801–804. IEEE, 2016.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, September 2003.

[Slo15] Peter Sloot. Model Execution: Event driven versus Time driven, 2015.

[Spa13] Petr Spacek. *Design and Implementation of a Reflective Component-Oriented Programming and Modeling Language*. PhD thesis, PhD thesis, Montpellier II University, Montpellier, France, 2013.

[SV02] Alberto Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, 2002.

[SVM01] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. 2001.

[Tei12] Jürgen Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012.

[TPF+16] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Instance migration in dynamic software update. 2016.

[VDMV14] Ken Vanherpen, Joachim Denil, Paul De Meulenaere, and Hans Vangheluwe. Design-Space Exploration in Model Driven Engineering. In *CEUR Workshop Proceedings: CMSEBA 2014*, pages 42–51, 2014.

[VF13] Ovidiu Vermesan and Peter Friess. *Converging Technologies for Smart Environments and Integrated Ecosystems*. River Publishers, 2013.

[VMD08] Yves Vanderperren, Wolfgang Mueller, and Wim Dehaene. Uml for electronic systems design: a comprehensive overview. *Design automation for embedded systems*, 12(4):261–292, 2008.

[WAU+08] Steve K Wood, David H Akehurst, Oleg Uzenkov, W Gareth J Howells, and Klaus D McDonald-Maier. A model-driven development approach to mapping uml state diagrams to synthesizable vhdl. *IEEE Transactions on Computers*, 57(10):1357–1371, 2008.

[WHMT08] Z. Wang, A. Herkersdorf, S. Merenda, and M. Tautschnig. A model driven development approach for implementing reactive systems in hardware. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 197–202, Sept 2008.

[Xil] Xilinx. Field Programmable Gate Array.

[YG02] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM Sigmod record*, 31(3):9–18, 2002.

[ZE+11] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[Zhu01] Jianwen Zhu. Metartl: Raising the abstraction level of rtl design. In *Proceedings of the conference on Design, automation and test in Europe*, pages 71–76. IEEE Press, 2001.

Software/FPGA Co-design for Edge-computing: Promoting Object-oriented Design
Co-conception Logiciel/FPGA pour Edge-computing: Promotion de la conception orientée objet

Cloud computing is often the most referenced computational model for Internet of Things. This model adopts a centralized architecture where all sensor data is stored and processed in a sole location. Despite of many advantages, this architecture suffers from a low scalability while the available data on the network is continuously increasing. It is worth noting that, currently, more than 50% internet connections are between things. This can lead to the reliability problem in realtime and latency-sensitive applications. Edge-computing, which is based on a decentralized architecture, is known as a solution for this emerging problem by: (1) reinforcing the equipment at the edge (things) of the network and (2) pushing the data processing to the edge.

Edge-centric computing requires sensors nodes with more software capability and processing power while, like any embedded systems, being constrained by energy consumption. Hybrid hardware systems consisting of FPGA and processor offer a good trade-off for this requirement. FPGAs are known to enable parallel and fast computation within a low energy budget. The coupled processor provides a flexible software environment for edge-centric nodes.

Applications design for such hybrid network/software/hardware (SW/HW) system always remains a challenged task. It covers a large domain of system level design from high level software to low-level hardware (FPGA). This results in a complex system design flow and involves the use of tools from different engineering domains. A common solution is to propose a heterogeneous design environment which combining/integrating these tools together. However, the heterogeneous nature of this approach can pose the reliability problem when it comes to data exchanges between tools.

Our motivation is to propose a homogeneous design methodology and environment for such system. We study the application of a modern design methodology, in particular object-oriented design (OOD), to the field of embedded systems. Our choice of OOD is motivated by the proven productivity of this methodology for the development of software systems. In the context of this thesis, we aim at using OOD to develop a homogeneous design environment for edge-centric systems. Our approach addresses three design concerns: (1) hardware design, where object-oriented principles and design patterns are used to improve the reusability, adaptability, and extensibility of the hardware system. (2) hardware / software co-design, for which we propose to use OOD to abstract the SW/HW integration and the communication that encourages the system modularity and flexibility. (3) middleware design for Edge Computing. We rely on a centralized development environment for distributed applications, while the middleware facilitates the integration of the peripheral nodes in the network, and allows automatic remote reconfiguration. Ultimately, our solution offers software flexibility for the implementation of complex distributed algorithms, complemented by the full exploitation of FPGAs performance. These are placed in the nodes, as close as possible to the acquisition of the data by the sensors, in order to deploy a first effective intensive treatment.

L'informatique en nuage (cloud computing) est souvent le modèle de calcul le plus référencé pour l'internet des objets (Internet of Things). Ce modèle adopte une architecture où toutes les données de capteur sont stockées et traitées de façon centralisée. Malgré de nombreux avantages, cette architecture souffre d'une faible évolutivité alors même que les données disponibles sur le réseau sont en constante augmentation. Il est à noter que, déjà actuellement, plus de 50 % des connexions sur Internet sont inter objets. Cela peut engendrer un problème de fiabilité dans les applications temps réel. Le calcul en périphérie (Edge computing) qui est basé sur une architecture décentralisée, est connue comme une solution pour ce problème émergent en: (1) renforçant l'équipement au bord du réseau et (2) poussant le traitement des données vers le bord.

Le calcul en périphérie nécessite des nœuds de capteurs dotés d'une plus grande capacité logicielle et d'une plus grande puissance de traitement, bien que contraints en consommation d'énergie. Les systèmes matériels hybrides constitués de FPGAs et de processeurs offrent un bon compromis pour cette exigence. Les FPGAs sont connus pour permettre des calculs exhibant un parallélisme spatial, aussi que pour leur rapidité, tout en respectant un budget énergétique limité. Coupler un processeur au FPGA pour former un noeud garantit de disposer d'un environnement logiciel flexible pour ce noeud.

La conception d'applications pour ce type de systèmes hybrides (réseau/logiciel/matériel) reste toujours une tâche difficile. Elle couvre un vaste domaine d'expertise allant du logiciel de haut niveau au matériel de bas niveau (FPGA). Il en résulte un flux de conception de système complexe, qui implique l'utilisation d'outils issus de différents domaines d'ingénierie. Une solution commune est de proposer un environnement de conception hétérogène qui combine/intègre l'ensemble de ces outils. Cependant, l'hétérogénéité intrinsèque de cette approche peut compromettre la fiabilité du système lors des échanges de données entre les outils.

L'objectif de ce travail est de proposer une méthodologie et un environnement de conception homogène pour un tel système. Cela repose sur l'application d'une méthodologie de conception moderne, en particulier la conception orientée objet (OOD), au domaine des systèmes embarqués. Notre choix de OOD est motivé par la productivité avérée de cette méthodologie pour le développement des systèmes logiciels. Dans le cadre de cette thèse, nous visons à utiliser OOD pour développer un environnement de conception homogène pour les systèmes de type Edge Computing. Notre approche aborde trois problèmes de conception: (1) la conception matérielle, où les principes orientés objet et les patrons de conception sont utilisés pour améliorer la réutilisation, l'adaptabilité et l'extensibilité du système matériel. (2) la co-conception matériel/logiciel, pour laquelle nous proposons une utilisation de OOD afin d'abstraire l'intégration et la communication entre matériel et logiciel, ce qui encourage la modularité et la flexibilité du système. (3) la conception d'un intergiciel pour l'Edge Computing. Ainsi il est possible de reposer sur un environnement de développement centralisé des applications distribuées, tandis ce que l'intergiciel facilite l'intégration des noeuds périphériques dans le réseau, et en permet la reconfiguration automatique à distance. Au final, notre solution offre une flexibilité logicielle pour la mise en oeuvre d'algorithmes distribués complexes, et permet la pleine exploitation des performances des FPGAs. Ceux ci sont placés dans les noeuds, au plus près de l'acquisition des données par les capteurs, pour déployer un premier traitement intensif efficace.

Keywords: IoT, Edge Computing, FPGA, Object Oriented Design, Distributed Objects
Mots clés: l'internet des objets, Calcul en Périphériques, FPGA,Conception orientée objet, Objets distribués